VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Optimalizace programovacího jazyka Kreatrix

Optimalization of programming language Kreatrix

2009 Stanislav Böhm

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 28. dubna 2009 .....

Děkuji vedoucímu diplomové práce ing. Marku Běhálkovi za rady, motivaci a možnost vytvořit tuto práci.

Abstrakt

Tato diplomová práce popisuje optimalizování implementace jazyka Kreatrix. Mezi hlavní optimalizace patří inline cache, JIT překladač a optimalizace zpracování bytekódu. Součástí práce je také přehled používaných optimalizačních technik u objektově orientovaných jazyků, popis implementace profilovacího modulu a srovnání rychlosti Kreatrix s implementacemi Pythonu, Ruby, PHP a Io.

Klíčová slova: Kreatrix, optimalizace, inline cache, JIT překladač

Abstract

This thesis focuses on the optimalization of implementation of the Kreatix language. Main optimalizations are inline cache, JIT compiler and optimalization of bytecode processing. The paper further contains overview of optimalization techniques for object oriented languages, description of profiling module implementation, and speed comparison of Kreatix and implementations of Python, Ruby, PHP and Io.

Keywords: Kreatrix, optimalization, inline cache, JIT compiler

Seznam použitých zkratek a symbolů

API	–	Application Interface
ASCII	–	American Standard Code for Information Interchange
CLOS	–	Common Lisp Object System
CPU	–	Central Processing Unit
HTML	–	Hyper Text Markup Language
JIT	–	Just-In-Time
JVM	–	Java Virtual Machine
LLVM	–	Low Level Virtual Machine
RISC	–	Reduced Instruction Set Computing
SGF	–	Smart Game Format
SVG	–	Scalable Vector Graphic
VM	–	Virtual Machine

Obsah

Seznam tabulek

Seznam obrázků

1 Úvod

Tato práce se zabývá tématem optimalizace implementace jazyka Kreatrix. Kreatrix je dynamický objektový jazyk, který vznikl jako má bakalářská práce. Po jejím dokončení a obhájení jsem ve svém volném čase na jazyku dále pracoval. Práce byla zaměřena především na funkčnost a ověření konceptu. Z tohoto důvodu byl kladen důraz na co nejjednodušší a snadno upravitelnou implementaci. Cílem této práce není obrátit vývoj přesně opačným směrem. Důležitým faktorem stále zůstává relativně jednoduchá implementace. Základní principy se již ale celkem stabilizovaly, takže má smysl úprava implementace s ohledem na výkon.

Cílem této práce je zefektivnit běh programů napsaných v jazyce Kreatrix. Cílem není upravit návrh jazyka tak, aby byl lépe optimalizovatelný. Uživatel jazyka by měl mít k dispozici stále všechny vlastnosti a abstrakci, jakou poskytovala implementace před zavedením optimalizací.

Vzhledem k tomu, že Kreatrix je dynamický jazyk s velkou mírou abstrakce, je obtížné využívat techniky optimalizace známé ze staticky kompilovaných jazyků. Kreatrix je objektový orientovaný jazyk s objekty založenými na prototypech a dědičností rozdílností, čímž se liší od většiny nejznámějších objektových jazyků. Z toho důvodu jsem musel některé optimalizační postupy upravit. Nejvýrazněji se to projevilo při implementaci inline cache.

Při zahájení práce jsem nejprve vytvořil sadu testů, abych mohl měřit zlepšení jednotlivých optimalizací (kapitola 5). Poté jsem se pomocí profileru snažil odhalit slabá místa a pomocí řady menších či větších změn odstranit nejvážnější výkonnostní problémy. Změny byly v rozsahu od drobných změn v základní knihovně až po větší změny v alokaci objektů. Většina těch významnějších změn je uvedena v této práci. Po těchto základních úpravách přišlo na řadu více experimentování a komplexnější optimalizace. Jedná se o inline cache (kapitola 7), rozšířené instrukce (kapitola 8.2) a JIT překladač (kapitola 9).

Většina optimalizací je zaměřena na rychlost provádění programu. Paměťovou náročnost jsem sledoval pouze druhotně, aby nedošlo k razantnímu nárůstu spotřeby paměti.

Součástí této práce je také modul `profiler` (kapitola 10), který slouží k profilování programů vytvořených v Kreatrix. Díky tomuto modulu má uživatel jazyka možnost snadněji najít výkonnostně slabá místa ve svém programu.

1.1 Označení verzí

Před vznikem této práce byla dokončena verze 0.10.0. Jako referenční verzi pro srovnání celkového přínosu používám verzi 0.10.1. Tato verze obsahuje oproti verzi 0.10.0 jen několik oprav a drobných vylepšení, které nemají vliv na výkon.

Verze 0.11.0 obsahuje všechny základní optimalizace. Jedná se především o optimalizace s cachováním objektů a mnoho dalších, poměrně jednoduchých úprav. Ve verzi 0.12.0 jsou zahrnuty dvě komplexnější optimalizace: rozšířené instrukce a inline cache. Výkonnostní srovnání mezi těmito větvemi je možné nalézt v závěru (kapitola 13).

2 Jazyk Kreatrix

Jazyk Kreatrix vznikl jako jazyk určený k implementaci serveru pro online hru. To ovlivnilo myšlenky a paradigmata, které byly při návrhu jazyka použity. Přesto však návrh vždy zohledňoval použití Kreatrix jako víceúčelového jazyka.

Snažil jsem si vzít dobré myšlenky z jazyků, které považuji za kvalitní a sloučit tyto nápady v nové podobě, v jaké jsem je nenacházel. Konkrétně se jedná o jazyk s prototypovými objekty, dědičností rozdílností a se syntaxí Smalltalku. Zároveň by měla mít implementace formu podobnou Pythonu, tedy malá VM s funkčností rozloženou do modulů a silnou podporou pro využití externích knihoven. Tedy bez „image“, kterou implementace Smalltalku často mají. Následující seznam obsahuje čtyři jazyky, kterými jsem byl nejvíce inspirován: Io[1], Self[2], Smalltalk[3] a Python[4].

Tato kapitola obsahuje krátký popis jazyka Kreatrix. Podrobnější popis sémantiky, syntaxe a historie vzniku Kreatrix je možné nalézt v mé bakalářské práci[1]. Další informace jsou dostupné také na stránkách projektu: `http://www.kreatrix.org`.

Stručný výčet charakteristik jazyka Kreatrix:

- Čistě objektově orientovaný jazyk (všechno je objekt)
- Objektový model založený na prototypech (neexistují třídy)
- Komunikace mezi objekty pouze pomocí zpráv
- Dynamicky typovaný
- Podpora výjimek
- Virtuální stroj implementován v C
- Podpora dynamického načítaní modulů
- Kompilace do bytekódu

2.1 Základní principy fungování Kreatrix

V této podkapitole je stručně rozebrán základní koncept objektů, slotů a zpráv v Kreatrix. Bližší detaily je možné nalézt na `http://www.kreatrix.org/documentation`.

2.1.1 Prototypový model

Jazyk Kreatrix patří mezi prototypové objektové jazyky. Díky tomu se pojetí objektů liší od jazyků, které jsou založené na třídách (například Smalltalk, Java, C++).

[1] `http://www.iolanguage.com`
[2] `http://research.sun.com/self/`
[3] `http://http://en.wikipedia.org/wiki/Smalltalk`
[4] `http://www.python.org/`

V jazycích se třídami jsou základem fungování dvě entity: *třída* a *objekt*. Třída drží společné vlastnosti objektů, které jsou instancemi dané třídy. Objekt samotný pak nese jen svá instanční data a ukazatel na svou třídu.

V jazycích s prototypy existuje jen jedna entita: *objekt*. Každý objekt je plně samostatný, bez závislosti na třídě. Objekt může obsahovat žádný, jeden nebo více rodičovských slotů. Každý rodičovský slot obsahuje referenci na další objekt. Pokud je objektu zaslána zpráva, kterou samotný objekt neumí zachytit, je využit rodičovský slot. V objektech, na které míří rodičovské sloty, se vyhledáváním do hloubky hledá objekt, který zprávu zachytí.

Pomocí tohoto mechanismu je možné vytvářet objekty, které drží společné chování pro ostatní. Oproti ukazateli na třídu je rodičovský slot volnější vazba a nijak neurčuje obsah samotného objektu. Rodičovské sloty mohou být za chodu kdykoli přidávány nebo odebírány.

2.1.2 Zasílání zpráv

Z pohledu programátora v Kreatrix obsahuje objekt tabulku slotů a seznam rodičovských slotů. Slot slouží jednak jako instanční proměnná, ale také jako místo pro uložení metod. Každý slot má své jméno a hodnotu (referenci na objekt).

Veškerá komunikace mezi objekty se děje pouze pomocí zasílání zpráv. Každá zpráva má jméno a seznam parametrů. Když dojde k zaslání zprávy objektu, nejprve proběhne vyhledání slotu. V objektu, který je cílem zprávy, se hledá slot se stejným jménem, jako má zpráva. Pokud je slot nalezen, je hledání u konce. Pokud nalezen není, pokračuje se rekurzivně hledáním v rodičovských slotech.

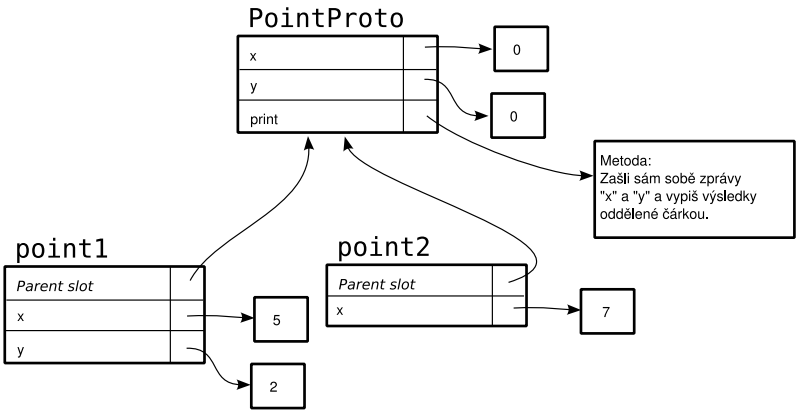
Pokud je slot nalezen, je vybrán objekt ze slotu. Je-li tento objekt aktivovatelný (například metoda), jsou předány parametry ze zprávy a objekt je spuštěn. Objekt, který je výsledkem spuštění aktivovatelného objektu, je také výsledkem zaslání zprávy. V případě, že objekt ve slotu není aktivovatelný, je výsledkem zprávy přímo tento objekt. Pokud není nalezen slot se jménem zprávy, je objektu zaslána zpráva `doesNotUnderstand` a pokud není nalezen ani tento slot, je vyhozena výjimka.

2.1.3 Dědičnost rozdílností

Nový objekt vzniká „naklonováním“ nějakého existujícího objektu. Nově vytvořený objekt nemá žádné vlastní sloty. Má pouze jeden rodičovský slot nastavený na objekt, ze kterého byl naklonován.

Protože je nový objekt prázdný, tak nemá jak zachytit zprávy, a proto hledání slotů vždy přejde do rodiče. Objekt proto reaguje na všechny zprávy stejně jako rodič. Pokud je do objektu vložen slot X, tak objekt bude opět reagovat na všechny zprávy jako rodič, kromě zprávy X.

Toto je podstatou dědičností rozdílností (differential inheritance). V objektu je zanesen rozdíl, kterým se liší od objektu v rodičovském slotu. Nově naklonovaný objekt má mít stejné chování jako rodič, neexistuje tedy žádný rozdíl, proto je nový objekt prázdný.



Obrázek 1: Ukázkové schéma objektů v Kreatrix

Pokud se má pak tento objekt lišit v reakci na zprávu X, tak je do něj zanesen nový slot, tedy rozdíl vůči rodiči.

V jazycích se třídami funguje podobný princip v dědění metod ve třídách. V jazycích s prototypy však toto může fungovat na úrovni obsahu samotných objektů. Toto chování je výhodné, pokud existuje velké množství klonů nějakého objektu, které se však od rodiče liší minimálně nebo vůbec. V případě jazyka s třídami má každá taková instance objektu všechny instanční proměnné bez ohledu na to, jestli se liší od inicializační hodnoty nebo ne. V Kreatrix budou nést tyto objekty pouze sloty se změnou.

Jazyk Kreatrix byl navržen pro implementaci herního světa a zde nachází tato vlastnost dobré uplatnění, protože v mapě se pohybuje často spousta podobných objektů, které se liší třeba jen jménem a pozicí. Přestože jsou ostatní vlastnosti většinu času na původní hodnotě, tato hodnota se může kdykoliv změnit.

2.1.4 Příklad

Tato podkapitola obsahuje ukázku, jak může vypadat jednoduchá struktura objektů v Kreatrix. Příklad se týká situace, která je zachycena na obrázku 1. Tři objekty jsou zobrazeny detailněji, u ostatních jsou zanedbány tabulky slotů a rodičovské sloty.

V příkladu je zobrazen objekt PointProto, který slouží jako prototyp pro body v dvou rozměrném prostoru. Objekty point1 a point2 jsou pak klony PointProto.

Pokud pošleme objektu point1 zprávu `x` nebo `y`, jsou příslušné sloty nalezeny hned v samotném objektu a jsou vráceny nalezené objekty. Pokud pošleme zprávu `print`, pak tato zpráva již v objektu point1 není zachycena a hledání pokračuje v rodičovském slotu. Zde je již slot `print` nalezen. Protože je ve slotu metoda, tedy aktivovatelný objekt, tak je metoda provedena a je vypsáno „5,2“.

Objekt point2 se liší tím, že v něm nedošlo ke změně souřadnice y, a proto nemá tento slot. Pokud je mu zaslána zpráva `y`, tak je výsledek nalezen v rodičovském objektu. Tedy po zprávě `print` reaguje point2 vypsáním „7,0“.

2.2 Ukázkové příklady kódu

Tato podkapitola obsahuje několik jednoduchých ukázek kódu. Bližší popis syntaxe a sémantiky je možné nalézt v mé bakalářské práci nebo na webových stránkách projektu.

2.2.1 Základní konstrukce

Několik jednoduchých příkladů pro použití cyklu a podmínky:

```
(x < 3) ifTrue: [ doSomething. ].

list foreach: [ :each | each println ].

10 to: 20 do: [ :i | i println ].
```

2.2.2 Jednoduché klonování objektu

V příkladu je vytvořen nový objekt `helloObject`, do kterého je vložena metoda `sayHello`.

```
helloObject = Object clone.

helloObject sayHello = {
    "Hello world!" println
}.

helloObject sayHello. // Poslání zprávy objektu helloObject
```

2.2.3 Výpočet faktoriálu

Rekurzivní výpočet faktoriálu:

```
Integer factorial = {
    self <= 1
        ifTrue: [ ^ 1 ]
        ifFalse: [ ^ self * (self - 1) factorial ]
}.

10 factorial println.
```

2.3 Closures

Tato kapitola si neklade za cíl podrobněji popisovat Kreatrix, spíše je jen rychlým přehledem základních principů. Přesto v této podkapitole detailněji popíši fungování closures, protože některé optimalizace úzce s jejich fungováním souvisejí.

Closure je funkce, která si pamatuje prostředí, ve kterém vznikla a při svém spuštění může využívat proměnné viditelné z tohoto prostředí. V syntaxi Kreatrix se podobně jako

ve Smalltalku vytváří closure pomocí hranatých závorek. Closure je objekt a lze ji tedy zasílat zprávy. Aktivování tohoto objektu je možné zasláním zprávy `value`, `value:`, `value:value:` atp, v závislosti na počtu parametrů. Objekt získaný vyhodnocením posledního výrazu v closure je vrácen jako výsledek.

```
test1 = { |closure string|
    // Vytvoří closure a uloží ji do proměnné.
    closure << [ string println ].

    string << "Hello".

    // Zavolání closure
    closure value. // Vypíše "Hello"

    string << "world".

    // Zavolání closure
    closure value. // Vypíše "world"
}.

test2 = {
    // Vytvoří a zavolá closure s dvěma parametry.
    [ :paraeter1 :parametr2 |
        parameter1 + parameter2 ] value: 3 value: 4.
}.
```

Closure v sobě nese dva ukazatele. První je ukazatel na CodeBlock obsahující bytekód, který se vykoná při spuštění closure. Druhý je pak ukazatel na aktivační záznam metody, ve které closure vznikla.

Při provádění je pak postupováno stejně jako při provádění metody, kód metody je také uložen ve formě CodeBlocku. Rozdíl je jen v tom, že kód metody je vykonáván v kontextu příjemce zprávy a kód closure je vykonáván v kontextu uloženého aktivačního záznamu.

Tato konstrukce je poměrně silný koncept a po vzoru Smalltalku jej Kreatrix velmi intenzivně využívá. Pomocí této konstrukce je realizováno také řízení toku kódu, tedy podmínky a cykly. Proto efektivita práce s tímto objektem má významný podíl na rychlosti celého virtuálního stroje. Optimalizaci těchto základních konstrukcí řeší rozšiřující instrukce (viz podkapitola 8.2).

2.4 Změny v Kreatrix oproti popisu v bakalářské práci

Mezi dokončením mé bakalářské práce a začátkem této práce proběhlo v Kreatrix velké množství změn. Ve většině případů se jednalo o rozšiřování funkcionality, jako například vznik nových modulů a nové možnosti virtuálního stroje. Několik úprav však změnilo

chování tak, že již neodpovídá popisu v bakalářské práci. Nejdůležitější změny jsou popsány v této podkapitole.

Detailnější seznam změn, které během této doby proběhly, je možné najít v souboru `changelog`[5]. Má bakalářská práce popisuje verzi 0.7. Verzí 0.11 počínaje začínají změny popsané v této práci.

2.4.1 Změna fungování lokálních slotů a aktivace metody

Tato změna je asi největší změnou, která od dokončení bakalářské práce proběhla. Změna zasahuje jak sémantiku, tak syntaxi a většina předchozího kódu musela být upravena. Přínosem je však jednodušší sémantika aktivace metody a zjednodušení práce s aktivačním záznamem, především zpřehlednění situace okolo lokálních slotů.

Změny budou ilustrovány na následujícím příkladu:

```
// Kód před verzí 0.8
obj method = {
    newObject = createNewObject.
    newObject doSomething.
    ^newObject.
}.

// Nový kód
obj method = { |newObject|
    newObject << createNewObject.
    newObject doSomething.
    ^newObject.
}.
```

Na první pohled je patrné, že nová syntaxe vyžaduje definovat lokální sloty na začátku metody (syntaxe převzata ze Smalltalku). Z toho důvodu je také místo aliasu pro vytvoření slotu „=“ použit alias „<<“, který slouží k aktualizaci slotů. Ke změně aliasu došlo, protože v době vykonávání metody již sloty existují. Není již tedy možné definovat lokální sloty průběžně za chodu. Pokud by však byla tato původní funkčnost vyžadována, stačí jen v lokálním slotu vytvořit pomocný objekt a do něj dynamicky sloty vkládat.

Alias „=“ v nových verzích představuje obyčejné zaslání zprávy `slot:set:`. Lokální sloty již nejsou nijak dotčeny. Proto je i stará verze metody přeložitelná novou verzí Kreatrix. Kód se ale chová jinak, oproti původnímu chování nezakládá lokální slot, ale ve výše zmíněném příkladu vytvoří nový slot `newObject` v objektu `obj`.

Tato změna umožnila odstranění objektu Activation (prototyp aktivačních záznamů). Spolu s tímto objektem mohly být odstraněny jeho metody, které musely mít speciální chování. Po této změně již rodičovský slot aktivačního záznamu míří na objekt, který je

[5]Tento soubor je součástí balíčku s Kreatrix. Obsah tohoto souboru je také dostupný na `http://www.kreatrix.org/changelog`.

příjemcem zprávy. Díky tomu je aktualizace lokálních slotů prováděna pouze pomocí jedné metody[6].

2.4.2 Další změny v syntaxi

Mezi další změny, které změnily syntaxi oproti popisu v bakalářské práci, patří změna provedená ve verzi 0.9. V této verzi byla zcela přepracována přední vrstva překladače. Důsledkem této změny je, že záporné číselné literály již nemusejí být kvůli unárnímu minusu v závorkách.

Další změnou v syntaxi je také odstranění znaku „|“ z povolených znaků pro binární zprávy. Z toho důvodu byly přejmenovány zprávy pro logické spojky „nebo“ a „a“ v objektech true a false. Zpráva || (dva znaky „pipe“) na \/ (zpětné lomítko a lomítko) a && na /\ (lomítko a zpětné lomítko).

[6]Ve verzi 0.11 a novějších je práce s lokálními sloty řešena pomocí speciálních instrukcí. Z pohledu z vnějšku je však chování, až na některé detaily, shodné s popisem, který je zde. Bližší informace o těchto instrukcích je možné nalézt v podkapitole 8.1.2.

3 Implementace virtuálního stroje Kreatrix

Optimalizace popisované v dalších kapitolách se často poměrně úzce týkají implementace virtuálního stroje. Proto je cílem této kapitoly stručně tuto implementaci popsat. Nejedná se o úplný popis nebo programátorskou příručku, ale pouze o popis základních principů s přihlédnutím k tématům popisovaným v dalších kapitolách.

3.1 Základní struktury v implementaci Kreatrix

Následující text popisuje základní struktury, které jsou používány v implementaci virtuálního stroje. Některá jména se objevují ve dvou podobách, s prefixem `Kx` a bez něj, například `CodeBlock` a `KxCodeBlock`. Prefix `Kx` používají struktury implementované v C. Bez prefixu je to jméno z pohledu API standardní knihovny Kreatrix, tak jak jej vidí uživatel jazyka. Jedná se tedy o jedno a to samé, jen z různého pohledu. Některé interní struktury virtuálního stroje nejsou uživateli jazyka dosažitelné, ty pak mají jen variantu s prefixem.

3.1.1 KxCore

Struktura `KxCore` reprezentuje instanci virtuálního stroje. Všechny ostatní struktury (s výjimkou `KxObjectExtension`) existují vždy v kontextu konkrétního `KxCore` a nelze použít například objekt z jedné instance `KxCore` v druhé. Pokud je Kreatrix použit jako samostatný jazyk, je vytvářen pouze jeden virtuální stroj.

3.1.2 KxObject

Instance této struktury reprezentuje objekt. Struktura obsahuje tabulku slotů, ukazatel na spojitý seznam parent slotů, počítadlo referencí a další údaje potřebné pro správu objektu.

Objekt může nést také vnitřní data. Informace o vnitřních datech je uživateli jazyka přímo nepřístupná a je možné na nich provádět operace pouze pomocí CFunction. Vnitřní data jsou například: číselná hodnota objektu reprezentující celé číslo, struktura s hashovací tabulkou v případě objektů Set a Dictionary, ukazatel na strukturu `FILE` v případě objektu reprezentující soubor.

Příkladem operace nad vnitřními daty může být objekt ve slotu `contains:` v objektu Set. Jedná se o CFunction, která přistupuje k vnitřním datům objektu (hashovací tabulce), a ověří, jestli je objekt v množině.

Struktury, které jsou přenášeny ve vnitřních datech objektů, mají ve většině případů příponu „Data“, viz dále `KxCFunctionData` a `KxCodeBlockData`. Samotná struktura `KxObject` má vždy stejnou podobu a velikost bez ohledu na to, jaká vnitřní data jsou přenášena. Další typy objektů, jako například `KxString` a `KxList`, jsou jen jiná jména pro `KxObject` pro zpřehlednění C kódu.

3.1.3 KxObjectExtension

Struktura `KxObject` obsahuje ukazatel na `KxObjectExtension`. Tato struktura obsahuje informace o tom, jak nakládat s vnitřními daty objektu. Jedná se například o operace při klonování objektu, rušení objektu nebo funkce pro spolupráci s garbage collectorem. Pokud objekt vnitřní data nemá, je tento ukazatel nastaven na `NULL`. Objekty se stejným typem vnitřních dat používají stejnou `KxObjectExtension`.

Ve funkcích v C je ověřován typ vnitřních dat podle ukazatele na tuto strukturu. Například objekty reprezentující řetězce ukazují na společnou strukturu. V této struktuře může být navíc zanesena dědičná hierarchie vnitřních dat pro flexibilnější typovou kontrolu. Tato vlastnost je například použita v modulu pro práci s GTK+[7].

Pokud je v této struktuře vyplněn ukazatel `activate`, stanou se objekty využívající tuto extension aktivovatelné. To znamená, že při aktivaci, místo vrácení samotného objektu, je zavolána funkce vyplněná v `activate`. Výsledkem aktivace je pak objekt, který vrátí tato funkce. Pomocí tohoto mechanizmu je pak možné realizovat metody jako objekty.

3.1.4 KxStack

Instance této struktury reprezentuje proces ve virtuálním stroji. Při normálním běhu si virtuální stroj vystačí pouze s jednou instancí `KxStack`. Více instancí této struktury je použito například v modulu spravující vlákna, kdy každé vlákno má vlastní `KxStack`. Další použití více `KxStack` nastává v případě, kdy v aktuálně běžícím `KxStack` dojde k výjimce a pro její zpracování je třeba volat další kód.

Nejdůležitější částí této struktury je zásobník pro aktuálně zpracovávané metody a zprávy. Slouží jako jeden z kořenů dostupnosti objektu v případě spuštění garbage collectoru.

3.1.5 KxCodeBlockData

Tato struktura představuje vnitřní data objektu `CodeBlock`. Tento objekt reprezentuje metodu a vzniká při načtení bytekódu do interpretru (viz podkapitola 3.2). Aktivací tohoto objektu dojde k provedení metody.

Struktura obsahuje seznam instrukcí, seznam literálů, tabulku zpráv, tabulku vnitřních bloků a počet parametrů a lokálních slotů. V rámci optimalizací, které jsou součástí této práce, přibylo do této struktury několik nových položek, například: inline cache a počítadlo pro rozpoznání hotspotu.

3.1.6 KxActivation

`KxActivation` vzniká pokaždé, když dojde k aktivaci objektu `CodeBlock`. Tato struktura udržuje informace o lokálních slotech, ukazatel na aktuální instrukci v bytekódu a další potřebné údaje pro provádění metody.

[7] `http://www.kreatrix.org/module-gtk`

3.1.7 CFunctionData

`CFunctionData` je struktura vnitřních dat objektu `CFunction`. Tento objekt je stejně jako CodeBlock aktivovatelný. Ovšem na rozdíl od CodeBlocku neprovádí bytekód, ale zavolá funkci implementovanou v jazyce C. Objekty typu CFunction jsou většinou použity u objektů s vnitřními daty. Pomocí CFunction jsou realizovány operace nad těmito daty, což je například případ objektů představující primitivní typy.

Další případ použití `CFunction` je u některých konstrukcí, které potřebují spolupracovat přímo s virtuálním strojem. Příkladem může být metoda ve slotu `throw:` v základním objektu, která způsobí vyhození výjimky. Další způsob použití `CFunction` je v případě metod, u kterých je potřeba maximální výkon.

Struktura `CFunctionData` obsahuje ukazatel na C funkci, počet parametrů a ukazatel na `KxObjectExtension`. Pokud není ukazatel na `KxObjectExtension` nastaven na `NULL`, musí mít objekt, nad kterým je CFunction aktivována, shodný ukazatel na tuto strukturu. Pokud není stejný, dojde k vyhození výjimky. Tento mechanizmus zabraňuje volání například funkce na zjištění délky řetězce nad objektem, který nemá jako vnitřní data řetězec. CFunction je objekt jako každý jiný a uživatel jazyka ho tedy může vložit i do jiných objektů.

3.2 Překlad do bytekódu

Každý zdrojový kód pro Kreatrix, aby mohl být prováděn virtuálním strojem, musí být nejprve přeložen do bytekódu. Bytekód je složen z instrukcí, které je virtuální stroj schopen vykonávat. Bytekód dále obsahuje další potřebné údaje pro sestavení CodeBlocku po načtení do virtuálního stroje.

Základní schéma překladu je zobrazeno na obrázku 2. Prvním krokem po načtení zdrojového kódu je lexikální analýza, ve které je zdrojový kód rozložen na tokeny. Při syntaktické analýze je sestaven strom bloků, který odpovídá stromu struktur CodeBlock po načtení do virtuálního stroje.

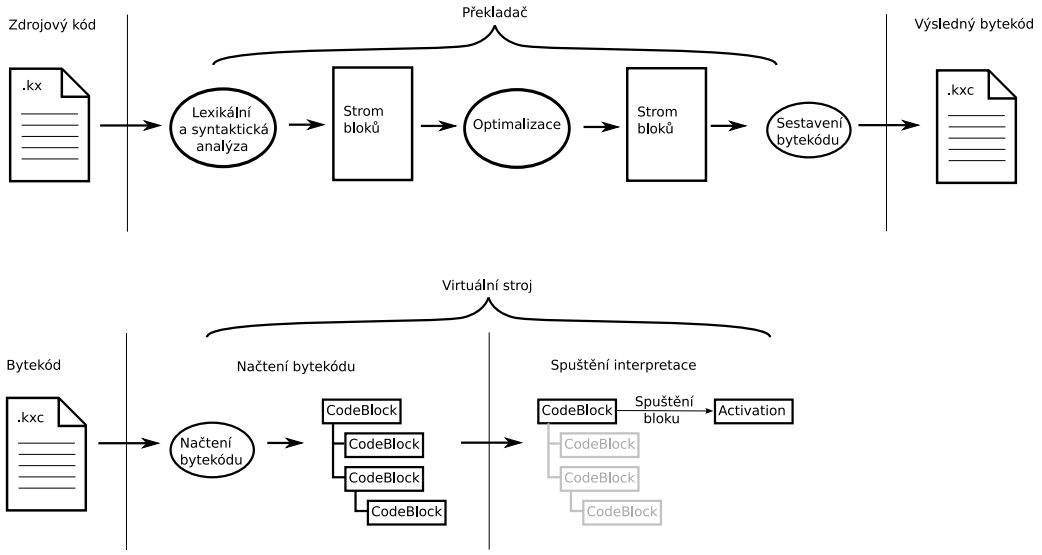
Po sestavení stromu bloků jsou provedeny další kroky týkající se optimalizací. Jedná se například o převod na rozšířené instrukce, toto je dále rozebráno v kapitole 8.2. Pro účely detekce prototypů jsou v této fázi hledány potenciální jména instančních slotů, tímto tématem se zabývá podkapitola 7.3.1.

Po provedení optimalizací je vygenerován bytekód. Toto generování je již triviální operací, protože instrukce bytekódu jsou poměrně abstraktní operace. Výsledný bytekód může být uložen do souboru nebo rovnou načten do virtuálního stroje.

3.3 Provádění instrukcí

Pokud je provedeno spuštění objektu CodeBlock, vytvoří se instance struktury KxActivation, která zachycuje stav běžící metody. Samotné provádění metody je pak realizováno vykonáváním instrukcí uložených v CodeBlocku.

Tyto instrukce operují nad zásobníkovým automatem. Při zaslání zprávy jsou parametry a případně cíl zprávy získány ze zásobníku. Výsledek po zaslání je pak opět vložen



Obrázek 2: Schéma překladu zdrojového kódu do bytekódu a jeho načtení do virtuálního stroje

na zásobník. Instrukční sada dále obsahuje instrukce pro manipulaci se zásobníkem, jako například „vlož literál na zásobník“ nebo „odeber objekt ze zásobníku“.

Popis všech instrukcí je možné nalézt v hlavičkovém souboru `kreatrix/src/compiler/kxinstr.h`. Bližší informace k formátu instrukcí jsou uvedeny v podkapitole 8.1.1.

4 Optimalizace objektově orientovaných jazyků

Objektově orientované jazyky přinesly nový způsob jak vytvářet programy. V mnoha případech přispívá objektové paradigma k lepší flexibilitě, znovupoužitelnosti a rozšiřitelnosti programu. Bohužel však programy využívající objektově orientovaného přístupu nejsou příliš dobře optimalizovatelné klasickými technikami, které jsou známé z procedurálních jazyků.

Pozdní vazba při volání metod v objektově orientovaných programech zabraňuje jednoduše sestavit graf volání funkcí, který mnoho statických optimalizací používá. Navíc objektově orientované programy jsou často sestaveny z mnoha malých metod, proto bývá volání metod mnohem častější, než volání funkcí v procedurálních programech.

V této kapitole bych rád shrnul nejznámější techniky, které se při optimalizaci objektových programů používají. Detailnější popisy je pak možné nalézt v další literatuře [2] [3] [4].

Optimalizace je možné rozdělit na dvě kategorie: statickou a dynamickou. Statická je prováděna v době překladu, dynamická pak v době běhu programu.

4.1 Statická optimalizace

Statická optimalizace má oproti dynamické výhodu v tom, že čas, který je při této optimalizaci spotřebován, se nijak neprojeví na době běhu programu. Může tedy provádět mnohem hlubší analýzu. Nevýhodou je, že nemá k dispozici běhové informace.

Základní myšlenkou mnoha optimalizací je snaha při zasílání zpráv určit množinu metod, která může být v daném místě spuštěna. V ideálním případě tato množina obsahuje jen jeden prvek. Překladač tedy rovnou ví, jaká metoda bude volána a může aplikovat další optimalizace, jako například vložení metody přímo do místa volání.

Taková situace bez složité detekce nastává například ve dvou případech: v místě vytváření nového objektu a v místě vložení literálu. Pokud v nějaké metodě je takto získán objekt a je mu zaslána zpráva, je možné přesně určit, jaká metoda bude spuštěna po zaslání zprávy.

4.1.1 Predikce třídy

Pokud nenastane jeden ze dvou výše zmíněných případů, nemusí být možné jednoznačně určit, která metoda má být zavolána. Jednou z optimalizačních technik může být predikce tříd. Je určeno několik typů (tříd), které jsou nejpravděpodobnějšími typy objektů, které budou daným místem procházet. Kód je v určitém místě rozvětven podle možných typů objektu. Protože je typ objektu v dané větvi již známý a je tedy známo, jaké metody budou volány po zaslání zprávy, mohou být provedeny další optimalizace.

Přidána je pak navíc větev, která pošle zprávy standardním způsobem v případě, že daný objekt není ani jedním z očekávaných typů. Pokud dokáže překladač spolehlivě určit konkrétní množinu typů daného objektu a ví, že za žádných okolností nemůže dorazit jiný typ, je tato větev vynechána.

Zjednodušenou variantou této optimalizace je „statická predikce třídy“. Pro určitá jména zpráv existuje předdefinovaná množina pravděpodobných příjemců zprávy. Například pro zprávu „+“ je možné očekávat, že příjemce zprávy je nejpravděpodobněji číslo. Odpadá tak nutnost složitější analýzy kódu. Nevýhodou je ztráta flexibility, kdy uživatel jazyka není schopen s vlastními konstrukcemi dosáhnout stejné efektivity, jako mají ty napevno zabudované.

Využití myšlenky predikce typu je v Kreatrix realizováno pomocí rozšířených instrukcí. Tímto tématem se zabývá kapitola 8.2.

4.1.2 Odesílací tabulky

Další možností statické optimalizace je použití odesílacích tabulek (dispatch tables)[5]. Ke každé třídě je sestavena tabulka s ukazateli na všechny metody, které mohou být pro danou třídu nalezeny. Každé zprávě je pak přiřazen pevný index do tabulky metod. Tento index pro zprávu daného jména je pro všechny třídy stejný. Při zaslání zprávy se tedy stačí jen podívat na patřičný index v odesílací tabulce.

Tato technika je užitečná především ve staticky typovaných jazycích, kde je pro každý typ známo, jaké zprávy je možné zasílat. Pro dynamicky typované jazyky by velikost tabulky musela být $m \times n$, kde m je počet všech tříd a n je počet všech možných jmen zpráv, protože každému objektu může být zaslána jakákoli zpráva. Z těchto důvodů není tato optimalizace v Kreatrix použita.

4.2 Dynamická optimalizace

Dynamická optimalizace má oproti statické optimalizaci výhodu, že může využívat informace o aktuálním běhu programu, protože je prováděna až po spuštění programu. Na druhou stranu z toho plyne její nevýhoda. Musí být dostatečně rychlá, aby její optimalizační efekt převážil nad režií s touto optimalizací spojenou.

Mezi dynamické optimalizace patří například JIT kompilace a inline cache. Princip JIT kompilace spočívá v kompilování metod do nativního kódu pro daný procesor v době běhu programu. Obecným popisem a implementací v Kreatrix se zabývá kapitola 9.

Inline cache je cache v místě odesílání zprávy. Při odeslání zprávy může program zapsat do cache výsledek vyhledávání. Při dalším průchodu daným místem může být cache využita a vyhledávání zprávy může být vynecháno. Výhodou této optimalizace je její přizpůsobivost skutečným podmínkám, které nastávají při vykonávání metod.

Sémantika Kreatrix, oproti jiným jazykům, přináší některé další problémy, které například jazyky založené na třídách nepotřebují řešit. Popis tohoto tématu lze nalézt v kapitole 7.

5 Výkonnostní testy

Před návrhem a implementací samotných optimalizací jsem jako první krok vytvořil sadu výkonnostních testů. Na těchto testech jsem pak prováděl pozorování a měřil vliv jednotlivých optimalizací. Využil jsem je také jako zdroj pro profilování, abych věděl, kterým směrem se mají optimalizace ubírat.

Výkonnostní testy jsem sestavil podle následujících kritérií: musí být neinteraktivní, deterministické, opakovaně spustitelné a při svém běhu by měly strávit co nejvíce času v kódu virtuálního stroje. Výsledný efekt testu by měl být nezávislý na verzi implementace[8]. Test by měl také trvat relativně dlouhou dobu, aby se minimalizovalo zkreslení doby běhu způsobené náhodnými vlivy (přepínáním procesů atp.).

5.1 Seznam testů

Následuje přehled testů, které byly použity pro měření vlivu mezi optimalizacemi. Tyto testy jsou pak dále v textu uváděny v grafech a tabulkách u jednotlivých optimalizací. Zdrojové kódy testů je možné nalézt v adresáři `benchmarks` v repozitáři nebo balíku s implementací jazyka Kreatrix.

- *t1_recursive* – Test rekurzivního volání. Test je převzat z The Computer Language Benchmarks Game[9].
- *t1_sort* – Test vygeneruje sestupný seznam čísel a vzestupně je setřídí. Test je zaměřen na volání Kreatrix kódu z C kódu. Třídění je prováděno C funkcí, která volá blok kódu pro informaci o porovnání.
- *t1_sum* – Test postupně ve smyčce sčítá čísla $1 \ldots 500000$. Cílem testu je změřit výkon v jednoduchých cyklech.
- *gc* – Test se zaměřuje na rychlost garbage collectoru. Test vygeneruje 100000 objektů a pak opakovaně pouští garbage collector.
- *mdispatch* – Test polymorfního chování.
- *ccounter* – Test spočítá počet písmen, číslic a mezer v souboru.
- *wcounter* – Test vypíše deset nejpočetnějších slov v souboru.
- *bigobject* – Test zasílání zpráv objektu, který obsahuje stovky slotů[10].
- *graph* – Test ze zadaného grafu vygeneruje kostru grafu a vypíše ji na standardní výstup.

[8] Problematická může být například operace nad všemi objekty ve virtuálním stroji. Efekt tohoto testu by byl závislý na verzi, protože mezi verzemi se například rozrůstá základní knihovna objektů. Z tohoto důvodu nebyl například zařazen mezi testy program pro generování dokumentace.

[9] `http://shootout.alioth.debian.org/`

[10] Tento test byl přidán až v průběhu této práce, proto se v počátečních výsledcích neobjevuje.

- *world* – Simulace pohybujících se objektů v mapě. Test vygeneruje několik náhodných map a několik desítek objektů v nich. Každý objekt se náhodně pohybuje a je testováno, zda-li nedochází ke kolizi s mapou nebo ostatními objekty. Jedná se o základní funkčnost, jakou by měl mít server obsluhující herní svět.
- *sgf_player* – Test načte čtyři záznamy hry Go ze souboru ve formátu SGF[11]. Tyto záznamy pak podle pravidel přehraje a vypíše finální pozici na desce.

5.2 Sběr výsledků testů

Pro sběr dat z výkonnostních testů jsem vytvořil několik jednoduchých nástrojů. Tyto nástroje jsou vytvořeny kombinací skriptů v shellu a v Kreatrix. Výsledky jsou pak prezentovány pomocí prostého textu, grafů ve formátu SVG a kompletního přehledu s tabulkami a grafy. Přehled je generován pomocí nástroje LaTeX[12]. Grafy jsou generovány pomocí nástroje R[13]. Všechny skripty vytvořené pro tento účel je možné nalézt spolu s testy v adresáři `benchmarks`.

5.2.1 Měření doby běhu testů

Měření rychlosti jednotlivých testů byla jedna z nejdůležitějších funkcí výkonnostních testů, aby bylo možné objektivně porovnat přínos jednotlivých optimalizací.

Všechny výsledky testů uvedené v této práci byly změřeny na tomto počítači:

- CPU – Intel Core2Duo E6750 @ 2.66GHz
- Velikost operační paměti – 2GB RAM
- OS – GNU/Linux 2.6.22-gentoo-r9

Kreatrix i implementace dalších testovaných jazyků byly přeloženy pomocí GCC 4.1.2. V průběhu vývoje byly systém a základní knihovny tohoto počítače aktualizovány. Po každé větší aktualizaci jsem provedl testy znovu, zda-li nedošlo ke změně ve výkonu, žádná se však neobjevila. Pro větší přesnost jsou však všechny výsledky testů v této práci změřeny znovu na poslední, výše uvedené, konfiguraci.

Měření bylo prováděno podle následující metodiky. Každý test je spuštěn desetkrát a hodnoty jsou zprůměrovány. Toto celé je zopakováno desetkrát a z deseti průměrů je vybrán ten nejlepší.

Naměřené časy jsou získány pomocí modulu `time`, který byl pro tento účel implementován. Změřený čas se vždy týká doby běhu hlavní části testu. Do doby běhu testu není započítávána inicializace a zrušení virtuálního stroje, ani případná inicializace a ukončení testu.

Pokud není uvedeno jinak, měření probíhala v základní konfiguraci virtuálního stroje s vypnutou podporou pro vlákna (parametr `--disable-threads` v `configure`).

[11] `http://www.red-bean.com/sgf/`
[12] `http://www.latex-project.org/`
[13] `http://www.r-project.org/`

5.2.2 Tabulky s výkonnostními testy

V dalších kapitolách této práce uvádím u popisu jednotlivých optimalizací tabulky s výkonnostními testy. Jedná se o naměřené rychlosti bez použití optimalizace (sloupec „Čas 1“) a s použitím optimalizace (sloupec „Čas 2“). Dále je uveden jejich rozdíl a rozdíl vyjádřený procentuálně vůči původnímu času. Záporná čísla rozdílů znamenají kratší čas běhu s optimalizací a tedy zlepšení, když je optimalizace zahrnuta. Všechny časy jsou uváděny v milisekundách.

5.2.3 Záznam práce virtuálního stroje

Pro detailnější rozbor běhu virtuálního stroje jsem přidal podporu pro vytváření záznamu vnitřního chování. Kreatrix přeložená s touto podporou je však o několik řádů pomalejší než při normální konfiguraci, proto je tato volba standardně vypnuta. Tuto vlastnost je možné zapnout pomocí přepínače `--enable-vm-log` v `configure`.

Pokud je podpora vytváření záznamu zapnuta, je po spuštění v aktuálním adresáři vytvořen soubor `kreatrix.log`. V tomto souboru každý řádek odpovídá jedné zaznamenané události. V řádku jsou jednotlivé položky odděleny dvojtečkou. První položka identifikuje typ záznamu dvoupísmenným kódem. Význam dalších položek záleží na typu události.

V současné implementaci jsou zaznamenávány tyto události:

- *Vytvoření a zánik objektu.* Doplňujícími údaji jsou adresa objektu, typ vnitřních dat a počet slotů v době zrušení objektu.
- *Alokace, realokace a dealokace paměti.* Položky záznamu pak obsahují: adresu alokované paměti, velikost alokace, jméno zdrojového souboru a číslo řádku, na kterém se nachází příkaz alokující paměť. Zaznamenávány jsou pouze alokace, o které explicitně žádá virtuální stroj, alokace, které provádí externí knihovny, nejsou zaznamenávány.
- *Zaslání zprávy*. V záznamu je uloženo jméno zprávy, adresa příjemce a počet parametrů.
- *Časová značka*. Časová značka je vkládána každou milisekundu, ve které proběhne alespoň jeden záznam.

Velikost výsledných logů v případě výkonnostních testů se pohybuje v desítkách až stovkách MB. V případě nejdelšího testu *world.kx* má log asi 230MB bez záznamu odeslaných zpráv. Se záznamem zpráv má 490MB.

Po provedení testu je log jednoduchým programem (napsaným v Kreatrix) rozebrán a jsou z něj vybrána data, která slouží pro generování grafů a tabulek. Zpracování logů všech výkonnostních testů zabere několik desítek minut.

Grafy jsou generovány pomocí programu R. Jako vhodný nástroj se tento program ukázal nejenom proto, že umožňuje snadno dávkově generovat grafy, ale také nemá potíže s množstvím dat. Pokusil jsem se zpracovat data také v programu Statgraphics[14]

[14] `http://www.statgraphics.com`

pro semestrální projekt do předmětu Statistika I. Některé tabulky však tento program nedokázal načíst z důvodu nedostatku paměti.

5.3 Profilování výkonnostních testů

Pro profilování jsem použil nástroj Valgrind[15]. Jedná se o kolekci nástrojů pro analýzu kódu pro ladění a profilování linuxových aplikací. Program je uvolněn pod licencí GPL a běží na těchto systémech: x86/Linux, AMD64/Linux, PPC32/Linux a PPC64/Linux. Velkou výhodou je, že program nemusí být pro Valgrind nijak speciálně přeložen. Nevýhodou je však poměrně velké zpomalení běhu programu.

Jako paměťový debugger jsem používal Valgrind již dříve, jeho součástí jsou také tři profilovací nástroje: Callgrind, Cachegrind a Massif. Při optimalizaci Kreatrix jsem využíval Callgrind a Massif.

5.3.1 Callgrind

Callgrind je jeden z profilovacích nástrojů Valgrindu. Callgrind dokáže sestavit graf volání funkcí. Za běhu programu sbírá data o jednolivých voláních, která jsou pak po ukončení programu zapsána do logu. Součástí Valgrindu je konzolový program, který dokáže tato data zpracovat.

Pro zpracování výsledných dat existuje také nástroj KCachegrind[16]. Jedná se KDE/Qt nástroj pro grafickou vizualizaci dat, která shromáždil Callgrind. Pomocí tabulek a různých grafů, které tento nástroj umí zobrazit, je mnohem snazší zorientovat se v profilovacích informacích.

5.3.2 Massif

Massif je paměťový profiler. Při běhu programu sleduje využití paměti. Výstupem je informace o tom, kolik bylo v různých okamžicích alokováno paměti. Pro některé časy je pak také zaznamenáno, které funkce paměť alokovaly.

Primárním cílem optimalizací Kreatrix bylo především zvýšení rychlosti. Massif jsem proto používal jen pro pro ověření, nedochází-li k nějakému razantnímu zvýšení paměťové náročnosti.

[15] `http://valgrind.org`
[16] `http://kcachegrind.sourceforge.net/cgi-bin/show.cgi/KcacheGrindIndex`

6 Optimalizace interpretru

6.1 Cache

V rámci provedených testů na verzi 0.10.1 se ukázalo, že velkou část běhu tráví program ve funkcích pro správu paměti, především `malloc` a `free`. Hlavním důvodem je alokace a dealokace aktivačních záznamů a objektů. V grafu na obrázku 3 je zobrazen graf alokace paměti v testu *world.kx* v Kreatrix 0.10.1. Na prvních třech pozicích je alokace objektu (56B), tabulka slotů (48B) a aktivační záznam (164B).

Základní myšlenka následujících optimalizací vychází z toho, že v nejčastějších případech alokace paměti je alokována paměť z omezené množiny velikostí. Navíc je alokovaná paměť často využívána jen po velmi krátkou dobu a následně uvolněna. Cílem tedy je tyto kusy paměti místo uvolnění shromažďovat do cache. K volání `free` dojde jen tehdy, je-li cache plná. Při žádosti o paměť je vybrána paměť z cache. Alokování pomocí `malloc` proběhne jen tehdy, je-li cache prázdná.

6.1.1 Cache pro paměť objektů

Při vytvoření objektu ve verzi 0.10.0 je `malloc` volán alespoň dvakrát, při alokaci paměti pro samotný objekt a při alokaci paměti pro tabulku slotů. Pokud má objekt více než jeden rodičovský slot, pak je pro každý další tento slot také alokována paměť.

Rodičovské sloty jsou uloženy ve zřetězeném seznamu. Pro první rodičovský slot není třeba samostatné alokace, protože je součástí struktury objektu. Protože však téměř všechny objekty obsahují právě jeden rodičovský slot[17], je tato příčina alokace velmi vzácná[18]. V následujícím textu se budu zabývat pouze prvníma dvěma případy alokace.

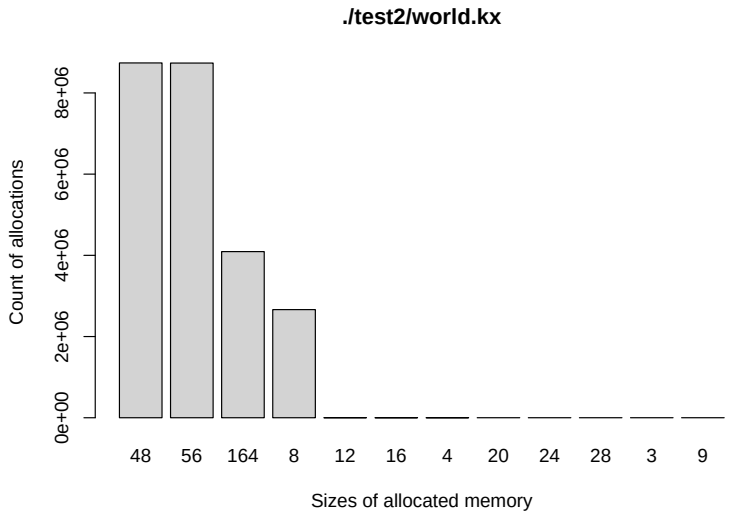
Struktura `KxObject` reprezentuje objekt. Tato struktura má stále stejnou velikost bez ohledu na to, jestli a jaká vnitřní data objekt nese. Pokud potřebuje objekt s vnitřními daty více paměti, než poskytuje základní struktura, tak je tato paměť alokována zvlášť. Alokací paměti pro vnitřní data se zde dále nebudu zabývat, protože není nijak centralizována a každý typ objektu si ji zajišťuje sám podle potřeby. Vnitřní data využívají některé základní objekty a objekty obalující externí struktury. „Normální“ objekt, vytvořený programátorem v Kreatrix, vnitřní data nenese.

Většina objektů má velmi krátkou životnost, ne více než několik zaslání zpráv (viz tabulka 1). Proto vzhledem k časté alokaci/dealokaci a stejně velkým úsekům paměti má smysl zavést cache nebo vlastní alokátor. Protože implementace cache je jednodušší a ukázala se jako vyhovující řešení, tak jsem nakonec k vytvoření vlastního alokátoru nepřistoupil.

Cache pro strukturu objektu je realizována jako zásobník v `KxCore`. Experimentálně se ukázalo, že velikost cache nad 50 se již nijak viditelně neprojevuje na zrychlení, proto je v následujících verzích velikost cache nastavena na tuto hodnotu. Výsledky výkonnostních testů jsou uvedeny v tabulce 2.

[17] Ve výkonnostních testech má právě jeden slot více než 99,9% objektů.

[18] Ve starších verzích byla alokována paměť pro každý rodičovský slot, včetně prvního. Úprava, která změnila chování tak, jak je popsáno výše, proběhla ještě před vznikem této práce.



Obrázek 3: Graf alokace paměti v průběhu testu *world.kx* v Kreatrix 0.10.1.

Test	Min	1. kvartil	Medián	Průměr	3. kvartil	Max
t1_recursive	0	1	3	2059	12	3669677
t1_sort	1	4502476	4503489	4460100	4504502	4504803
t1_sum	0	1	1	507	1	500299
gc	1	52061	102588	102580	153115	201908
mdispatch	1	1	5	5278	21	5863244
ccounter	0	1	2	1449	7	597543
wcounter	0	0	7	2691	1690	3266082
graph	0	1	1	2215	4	634582
world	0	1	4	22556	11	15933283
sgfplayer	0	1	10	13097	113	3775613

Tabulka 1: Počet zaslaných zpráv od vzniku po zánik objektu ve výkonnostních testech.

Benchmark	Čas 1 (ms)	Čas 2 (ms)	Rozdíl	Rozdíl v %
gc	5519	5565	46	0,833%
mdispatch	2623	2405	-218	-8,311%
t1_recursive	1767	1581	-186	-10,526%
t1_sort	2724	2529	-195	-7,158%
t1_sum	543	484	-59	-10,865%
ccounter	291	257	-34	-11,683%
graph	314	282	-32	-10,191%
wcounter	856	804	-52	-6,074%
world	5582	5230	-352	-6,305%
sgfplayer	1332	1248	-84	-6,306%

Tabulka 2: Výsledky výkonnostních testů s cachováním paměti objektů.

Alokace paměti pro sloty je problematičtější, protože není dopředu známo, kolik slotů bude daný objekt mít. Objekt vždy vzniká s nulovým počtem slotů a sloty jsou do něj přidávány dynamicky za chodu. Zkusil jsem implementovat několik různých řešení uložení slotů. Jako nejlepší řešení se z hlediska výkonu ukázaly být hashovací tabulky. Tato problematika je dále popsána v podkapitole 6.2.

6.1.2 Cache aktivačních záznamů

Další funkce, která způsobovala časté volání `malloc`, byla alokace aktivačního záznamu. Aktivační záznam je vytvořen při spuštění metody a nese informace potřebné pro provádění metody, např: objekt, nad kterým je metoda vykonávána, ukazatel na zprávu, která metodu spustila, atp.

V Kreatrix ale nelze aktivační záznamy jednoduše alokovat na zásobníku (jako například v jazyce C). Důvodem je, že aktivační záznam nemusí být dealokován po ukončení metody. Tato situace nastane, pokud je v metodě vytvořena closure, která nezanikne po ukončení metody. Protože closures využívají aktivační záznam metody, ve kterém jsou vytvořeny, je možné dealokovat aktivační záznam, až již neexistuje žádná closure pro danou aktivaci metody. Tato častá alokace byla také vyřešena pomocí cache. Testování na výkonnostních testech ukázalo, že již malé hodnoty velikosti cache eliminují téměř všechna volání `malloc` z tohoto důvodu.

Optimalizace aktivačních záznamů by mohla dále pokračovat. Může být využito například vlastnosti, že aktivační záznamy, které jsou potřeba i po skončení metody, jsou relativně vzácné. Výsledky z profileru po zavedení cache však byly více než dostačující, tak jsem od dalších optimalizací v této oblasti upustil.

Výsledky výkonnostních testů jsou uvedeny v tabulce 3.

6.1.3 Cache malých celých čísel a znaků

Dalším vhodným místem pro optimalizace je znovupoužívání objektů, jejichž vnitřní data jsou neměnná (immutable) a které lze efektivně v cachi vyhledávat. Takovými ob-

Benchmark	Čas 1 (ms)	Čas 2 (ms)	Rozdíl	Rozdíl v %
gc	2171	2172	1	0,046%
mdispatch	2110	1900	-210	-9,952%
t1_recursive	1392	1314	-78	-5,603%
t1_sort	2564	2254	-310	-12,090%
t1_sum	423	405	-18	-4,255%
ccounter	225	209	-16	-7,111%
graph	260	235	-25	-9,615%
wcounter	773	710	-63	-8,150%
world	5059	4757	-302	-5,969%
sgfplayer	1189	1118	-71	-5,971%

Tabulka 3: Výsledky výkonnostních testů s cachováním aktivačních záznamů.

jekty jsou například objekty reprezentující znak. Daný objekt bude vždy reprezentovat konkrétní znak, vnitřní data jsou neměnná. Takže referenci na takový objekt mohou držet další objekty bez obavy z toho, že by nějaká další strana, která také drží referenci, mohla změnit význam objektu.

Znaky je možné efektivně ukládat v cachi, protože je reprezentuje relativně malá číselná hodnota. Tedy je možné použít hodnotu znaku jako index do tabulky. [19]

Mezi neměnné objekty patří také některé další základní typy objektů: celá čísla, čísla s plovoucí řadovou čárkou, řetězce a symboly. Pro malá celá čísla je zavedena tabulka cache podobně jako pro znaky. Řetězce a čísla s plovoucí řadovou čárkou nejsou cachovány, protože u nich nelze najít malou omezenou množinu často používaných objektů. Symboly již ze své podstaty není třeba ukládat do cache[20].

V případě znaků je cachováno všech 127 znaků ASCII tabulky. V případě celých čísel cache obsahuje čísla -10 až 100. Výsledky výkonnostních testů jsou uvedeny v tabulce 4.

6.2 Vyhledávání ve slotech pomocí hashovací tabulky

Virtuální stroj stráví velkou část výpočetního času ve funkci pro hledání slotů v objektech[21]. Řešení tohoto problému probíhalo dvěma způsoby: eliminování vyhledávání slotů a zrychlení samotného vyhledávání. První přístup je realizován pomocí inline cache, které je věnována kapitola 7. V této podkapitole popíši druhý způsob.

Původní implementace uchovává tabulku slotů jako obyčejné pole. Při vyhledání slotů je pak toto pole sekvenčně procházeno. Toto řešení jsem zvolil kvůli snadné imple-

[19] Současné verze Kreatrix používají 8-bitové kódování. Podpora multibyte kódování je plánována, ale v době psaní této práce zatím není implementována. Úprava této cache pro podporu multibyte kódování by ale neměla být příliš problematická. V takovém případě by ale nebyly uloženy v cachi všechny znaky.

[20] Z důvodu efektivního porovnávání existuje symbol se stejným jménem v paměti vždy jen jednou. Stačí jen porovnat ukazatel místo celého jména symbolu.

[21] Ve verzi 0.11.1 tráví virtuální stroj zhruba 15% až 35% celkového času vyhledáváním slotů. Měřeno při provádění výkonnostních testů.

Benchmark	Čas 1 (ms)	Čas 2 (ms)	Rozdíl	Rozdíl v %
bigobject	2581	2586	5	0,193%
gc	1778	1791	13	0,731%
mdispatch	1294	1220	-74	-5,718%
t1_recursive	825	789	-36	-4,363%
t1_sort	1098	1118	20	1,821%
t1_sum	203	215	12	5,911%
ccounter	144	142	-2	-1,388%
graph	157	147	-10	-6,369%
wcounter	410	409	-1	-0,243%
world	2962	2868	-94	-3,173%
sgfplayer	708	689	-19	-2,683%

Tabulka 4: Výsledky výkonnostních testů s cachováním objektů pro malá celá čísla a znaky.

mentaci. Také ale proto, že nebylo jasné, jaká bude efektivní datová struktura pro uložení slotů. Proto, abych se vyhnul předčasné optimalizaci, zvolil jsem to nejjednodušší řešení.

Při problému uložení slotů se setkává několik špatně skloubitelných požadavků. Nejčastější operací nad sloty v objektu je vyhledávání, takže tato operace musí být co nejrychlejší. Sloty jsou ovšem také poměrně často přidávány. Objektů je v celém systému velké množství, proto se velikost tabulky slotů značně promítne do paměťové náročnosti programu. Využití paměti pro uložení slotů musí být tedy poměrně efektivní. Většina objektů má pouze několik slotů, velká část objektů nemá sloty vůbec. Objekty držící společné chování mají obvykle maximálně několik desítek slotů. Systém by však měl být připraven i na objekty s tisícovkami slotů[22].

Nakonec jsem zvolil řešení pomocí hashovacích tabulek. Implementoval jsem dva algoritmy pro hledání v hashovací tabulce: Kukačkové hashování[6] a Lineární sondáž[7]. Obě hashovací metody testu `bigobject` hodně pomohly, protože podstatou testu je zasílaní zpráv objektu, který obsahuje velké množství slotů. U ostatních testů je už přínos minimální, často negativní.

Při sloučení těchto větví s optimalizací pomocí inline cache je výsledek ještě více nepříznivý. Při použití inline cache je většina testů výrazně pomalejší než sekvenční prohledávání. Problém nastává v objektech s malým množstvím slotů. V takovém objektu je pak sekvenční vyhledání rychlé a dává lepší výsledky než použití hashovací tabulky. Zároveň použití inline cache typicky eliminuje opakované vyhledávání v objektech s více sloty, takže se tento propad mezi původním prohledáváním a hashovacími tabulkami ještě více prohloubí.

Jako nejlepší řešení se proto ukázal kompromis mezi oběma řešeními. Malé objekty jsou prohledávány sekvenčně a od určité velikosti je použita hashovací tabulka. Jednodušší algoritmus lineární sondáže byl ve výsledcích mírně lepší než kukačkové hasho-

[22] Například objekty, které drží konstanty v modulu GTK, mají řádově stovky slotů.

Benchmark	Čas 1 (ms)	Čas 2 (ms)	Rozdíl	Rozdíl v %
bigobject	2502	984	-1518	-60,671%
gc	1795	2292	497	27,688%
mdispatch	1246	1191	-55	-4,414%
t1_recursive	781	744	-37	-4,737%
t1_sort	1057	1120	63	5,960%
t1_sum	207	194	-13	-6,280%
ccounter	141	137	-4	-2,836%
graph	146	150	4	2,739%
wcounter	401	414	13	3,241%
world	2887	2687	-200	-6,927%
sgfplayer	694	653	-41	-5,907%

Tabulka 5: Výsledky výkonnostních testů při použití sekvenčního vyhledávání ve slotech (Čas 1) a použití finální podoby hashovací tabulky (Čas 2).

vání. Z tohoto důvodu je proto ve finalní verzi použita lineární sondáž. Výkonnostní porovnání je možné nalézt v tabulce 5. Některé testy jsou sice oproti původní implementaci mírně pomalejší, ale ve většině případů došlo ke zrychlení. Test `bigobject` je po této změně dvaapůlkrát rychlejší.

6.3 Volitelná podpora více virtuálních strojů

Implementace jazyka Kreatrix byla od počátku navržená jako vložitelná do jiného prostředí. Tedy je možné použít Kreatrix jako skriptovací jazyk pro jinou aplikaci. Proto implementace již od prvních verzí podporuje možnost mít více instancí virtuálního stroje v rámci jednoho programu.

Nevýhodou tohoto řešení je, že součástí každého objektu musí být ukazatel na `KxCore`. `KxCore` je struktura reprezentující instanci virtuálního stroje Kreatrix. Možnost využití více virtuálních strojů v jednom programu se v současné době však nijak nevyužívá. Navíc při použití Kreatrix jako samostatného jazyka nemá ani příliš opodstatnění.

Protože však nechci tuto možnost z implementace úplně odstranit, rozhodl jsem se pro řešení, ve kterém je možné zkompilovat Kreatrix volitelně s podporou, nebo bez podpory této vlastnosti. V `configure` je nyní možné pomocí volby `--enable-multistate` tuto podporu zapnout. Vzhledem k současné situaci je tato volba standardně vypnuta. Pokud je podpora vypnuta, existuje pouze jedno globální `KxCore`, kterému patří všechny vzniklé objekty. Ze struktury `KxObject` je ukazatel na `KxCore` odstraněn.

Po provedení výkonnostních testů došlo k malému zvýšení rychlosti u většiny testů. Znatelný rychlostní rozdíl se objevil jen u testu GC, kdy se příznivě projevila nižší velikost objektů.

7 Inline cache

Inline cache je jedna z metod optimalizace vyhledávání slotů v objektech. Jedná se o dynamickou optimalizaci, která využívá informace získávané v době běhu programu. Přestože se jedná o jednu z dalších optimalizací interpretru, vyčlenil jsem jí samostatnou kapitolu, protože se jedná o rozsáhlejší tématiku.

7.1 Základní princip inline cache

Z důvodu snadnějšího vysvětlení budu v této podkapitole popisovat použití inline cache v dynamicky typovaném jazyku, který má objekty založené na třídách (například Smalltalk). Sémantika Kreatrix přináší pro inline cache některé další problémy. Těmto problémům a jejich řešení jsou pak věnovány další dvě podkapitoly.

Ve Smalltalku i Kreatrix je zasílání zpráv jediným prostředkem, jak mohou objekty mezi sebou komunikovat. Když dojde k zaslání zprávy, musí být nejprve zjištěno, jaká metoda se má spustit. Virtuální stroj v normálním (neoptimalizovaném) případě musí provést vyhledání metody v hierarchii tříd. Toto vyhledání představuje nezanedbatelnou režii, která musí být vykonána při každém zaslání zprávy.

Základní myšlenka předpokládá, že v mnoha případech je zpráva zasílána jen velmi omezené množině typů. Virtuální stroj si pak u konkrétní instrukce, která zasílá zprávu, poznamená do cache, jakému typu objektu je zpráva zaslána a jaká metoda byla nalezena. V případě, že vykonávání programu dorazí znova k této instrukci, je nejprve ověřeno, jestli příjemce zprávy není stejného typu jako v minulém vykonání. Pokud je tomu tak, pak je rovnou známa metoda díky uchovaným informacím z minulého vykonání. V takovém případě odpadá celá režie vyhledávání zprávy. Pokud typ objektu neodpovídá typu v cachi, je provedeno normální vyhledání a výsledek uložen do cache.

Následuje ukázka na konkrétním případě. Metoda je napsána v jazyce Smalltalk[23]:

```
pridejJednickuK: parametr
    ^ parametr + 1
```

Tato metoda pošle svému parametru binární zprávu „+“ s parametrem 1. Pohledem na tělo metody je vidět, že očekávaným parametrem metody bude nejspíše nějaké číslo. Toto ovšem překladač bez nějaké komplexní analýzy programu nemůže s jistotou vědět.

Pokud by tato metoda používala inline cache a bylo by jí podle očekávání jako parametr dosazováno například pouze celé číslo, dojde z pohledu inline cache k ideální situaci. Po prvním spuštění metody bude do cache uložena metoda pro sčítání celých čísel. Všechna další taková spuštění již budou pouze využívat cache, bez nutnosti vyhledávat metodu v hierarchii tříd. Pokud by došlo k tomu, že parametrem by byl jiný objekt než celé číslo, dojde k normálnímu vyhledání metody a aktualizaci cache.

[23] Vzhledem k předchozímu popisu chování inline cache na jazycích s třídami volím i příklad v takovém jazyce. Přepis metody do Kreatrix:
`pridejJednickuK: = { :parametr | ^ parametr + 1 }`

Největší výhodou je flexibilita této optimalizace, protože vychází ze skutečného běhu aplikace. Pokud budou v předchozím příkladu do metody dosazovány objekty reprezentující číslo v plovoucí řadové čárce nebo jiné objekty implementující vlastní „+“, tak dojde k využití cache. Cache se naplní takovým typem objektu, jaký bude reálně metodou procházet.

Tato cache není výhodná pro místa, kde je zpráva zasílána větší množině typů objektů. V následujícím textu se předpokládá, že cache si pro konkrétní místo v bytekódu umí pamatovat jen jeden objekt. Proto, pokud se pravidelně střídají dva a více typů objektů, je cache neúčinná a přináší pouze režii navíc. Tento problém řeší polymorfní inline cache, která je obsahem kapitoly 7.7.

Další nevýhodou je vyšší spotřeba paměti a zbytečné cachování, pokud je metoda s inline cachí prováděna jen jednou nebo velmi zřídka. Tyto problémy se však dají minimalizovat tím, že je inline cache aktivována pouze v metodách, ve kterých tráví program nejvíce času. Detekce těchto míst v Kreatrix je řešena v kapitole 7.5.

7.2 Problémy použití inline cache v Kreatrix

Inline cache nelze v jazycích, které mají podobnou sémantiku jako Kreatrix, použít přímo tak, jak je popsáno výše. Problém se týká toho kroku, kdy je do inline cache uložena informace o tom, že v objektech typu T vyhledání metody pro danou zprávu vrací objekt X. Při dalším průchodu je pak ověřeno, jestli cíl zprávy je typu T a může použít cache.

Problém se týká ověření, zda-li je nějaký objekt typu T. Ve výše popsaném jazyce s třídami si stačí uložit danou třídu a testovat, zda-li objekt je instancí této třídy. V Kreatrix však třídy neexistují. Každý objekt je unikátní a rodičovský slot nelze použít ve stejném významu jako ukazatel instance na třídu. Rodičovský slot sjednocuje vazbu „je instancí“ a „dědí z“ do jedné.

Používat proto přímo rodičovský slot pro detekci typu není možné. Na rozdíl od ukazatele na třídu nelze na základě shodnosti rodičovských slotů u dvou objektů říci, jestli vyhledání ve slotech vrátí stejný výsledek. Každý objekt může nést vlastní sloty, vyhledání může rovnou nalézt výsledek v samotném objektu a k použití rodičovského slotu vůbec nemusí dojít.

Situace je z pohledu inline cache ještě navíc komplikována tím, že sloty unifikují instanční proměnné a místa, ve kterých jsou uložené metody. Takže to, že objekt má vlastní sloty, je velmi běžné.

Pokud je ve třídě změněna metoda, musí se provést nějaké ověření, zda-li není třeba aktualizovat inline cache a případně nekonzistentní místa nalézt a opravit. V normálním programu není změna metod příliš častá operace, proto je tato režie přijatelná. V případě Kreatrix však nastává problém s těmi sloty, které slouží jako instanční proměnné. Ty jsou velmi často vytvářeny nebo měněny, proto u takových slotů nemá inline cache smysl.

Rozdělení na sloty, které jsou instanční a ty, které drží obecné vlastnosti a chování, je však jen umělé rozdělení pro účely tohoto textu. V Kreatrix existuje pouze jeden druh slotu a programátor nijak neurčuje jeho typ. Proto není pro virtuální stroj jednoduché rozhodnout, které sloty optimalizovat a u kterých to nemá smysl.

7.3 Detekce prototypů a instancí

V předchozí podkapitole byly popsány problémy, kvůli kterým nelze implementovat inline cache v Kreatrix stejným způsobem jako u jazyků založených na třídách. Tato podkapitola popisuje řešení, jak tyto problémy odstranit tak, aby mohla být inline cache použita. Řešení poskytuje metodu, jak rychle určit, zda-li je objekt nějakého typu pro účely inline cache. Součástí řešení je také způsob, jak detekovat sloty, které jsou trvalého charakteru a ty, které se mohou často měnit a nemá je tedy smysl takto optimalizovat.

Základem mého řešení je předpoklad, že se v programu vyskytují dva druhy objektů. Jeden druh jsou objekty, které drží společné chování pro ostatní a mění se relativně málo. Druhý druh objektů pak obsahuje sloty, ve kterých je udržován vlastní stav objektu a tyto sloty se mohou měnit často. Tyto objekty pro svou funkčnost využívají přes rodičovský slot první druh objektů.

Nemusí to být nutně model na způsob tříd a instancí. Sémantika Kreatrix umožňuje poměrně flexibilní pohled na objekty. Mé řešení pro inline cache také počítá s možností využití dědičnosti rozdílností, tedy toho, že v objektech vznikají sloty postupně, jak se klon liší od prototypu. Pokud však uživatel přijde s naprosto odlišným přístupem na použití objektů a kde většina objektů nesplňuje rozdělení na tyto dvě skupiny, pak bude asi stát za zvážení vypnutí této optimalizace. Místa, ve kterých může nastat problém, jsou popsána dále.

Navrhl jsem několik způsobů, jak řešit tuto detekci. Jeden z neúspěšných pokusů je například popsán v kapitole 7.8. Dále popsané řešení je to úspěšné, které bylo začleněno do hlavního stromu.

V tomto řešení je základní struktura objektu rozšířena o p-typ[24] a referenci na profil. P-typ může nabývat hodnoty *prototyp* nebo *instance*. V případech, kdy budu v následujícím textu mluvit o instancích a prototypech, mám na mysli právě označení objektu pomocí p-typu. Z pohledu virtuálního stroje však stále existuje jen jeden druh objektu, tato informace slouží pouze pro účely inline cache a na další funkce nemá žádný vliv.

Prozatím se omezím na případ, kdy má objekt právě jeden rodičovský slot, situaci s více rodičovskými sloty popíši dále. Cílem je dosáhnout následujícího stavu (hlavní podmínka):

Pro všechny objekty, které jsou označeny jako instance, platí, že mají profil stejný, jako objekt v jejich rodičovském slotu a obsahují pouze jména slotů uvedená v profilu jako instanční sloty.

Jinak řečeno, pokud je objektu označeném jako instance zasílána zpráva, která není uvedena v profilu, tak je možné hledání začít přímo v rodiči. Pokud je tato podmínka splněna, pak je již jednoduché použít inline cache.

Inline cache je možné použít, pokud je objekt, kterému je zpráva určena, označen jako instance a v jeho profilu není jméno zprávy, jaká je v daném místě zasílána. Do cache se při prvním průchodu uloží profil a výsledek vyhledávání ve slotech. Při dalším průchodu se pak ověří, jestli profil příjemce zprávy je shodný s profilem v inline cache a tedy jestli se může cache použít. Ukazatel na profil tedy plní stejnou funkci jako ukazatel na třídu ve výše popsaném případě inline cache v jazycích s třídami.

[24] Toto jméno je zvoleno z historických důvodů a ponechal jsem ho, aby nedošlo k záměně s pojmem „typ objektu“.

Konkrétní rozbor využití informací p-typu a profilu při použití inline cache je popsán dále v kapitole 7.4. V následujících odstavcích se budu věnovat tomu, jak je získávána informace o p-typu a profilu a jak je udržována jejich konzistence.

7.3.1 Jména zpráv v profilu

Aby předchozí mechanizmus mohl korektně fungovat, je potřeba, aby profil obsahoval správná jména zpráv. Tato jména jsou jména slotů, která může obsahovat objekt označený jako instance. Jsou to sloty, které může takový objekt měnit, aniž by se tím porušila konzistence cache. Základní otázkou tedy je, jak určit, které sloty mají být do profilu zařazeny.

Problém, jestli nějaký slot patří do profilu, spočívá v tom, zda-li se chová daný slot jako instanční nebo jestli spíše drží obecné chování. Znovu připomínám, že v Kreatrix existuje pouze *jeden* typ slotu. Dělení na instanční a neinstanční sloty je pouze pojmenování slotu podle toho, jak je v programu se slotem nakládáno. Programátor nijak explicitně tuto informaci do programu nevkládá. Program může být vykonán i bez tohoto rozlišování slotů, jedná se o informaci pouze pro použití s touto optimalizací.

Následující postup je v principu metoda, jak „odhadnout“, jestli se daný slot bude chovat spíše jako instanční nebo jako neinstanční. Pokud jako instanční, pak je zařazen do profilu.

Současná implementace takto detekuje instanční slot ve dvou případech. První případ nastává, pokud je použit čtecí/zapisovací pár[25]. Tato konstrukce vkládá do objektu jednoduchou metodu, která aktualizuje daný slot.

Příklad:

```
MyPoint = Object clone do: {
    x <- 0.
    y <- 0.

    asString = { ^ "(", x, ",", y, ")" }.
}
```

Objekt MyPoint bude obsahovat 5 slotů: `x`, `x:`, `y`, `y:` a `asString`. Sloty `x` a `y` budou vloženy do profilu v objektu MyPoint, protože k nim existuje zapisovací metoda. V tomto příkladě se jedná o metody uložené ve slotech `x:` a `y:`. Protože `<-` vždy vytváří zapisovací metodu, je při použití této konstrukce čtecí slot označen jako instanční.

Druhý způsob detekce instančního slotu se týká tohoto příkladu:

```
MyPoint = Object clone do: {
    x = 0.
    y = 0.

    setX: = { :param | doSomething. x = param }.
```

[25] `http://www.kreatrix.org/rwpair`

```
    setY: = { :param | y = param }.

    asString = { ^ "(", x, ",", y, ")" }.
}
```

a nebo tohoto příkladu:

```
import: "time".

MyObject = Object clone do: {

   /* Zpráva "init" je objektu automaticky
      zaslána ihned po naklonování */
   init = {
       initTime = time now.
   }
}
```

V prvním příkladu je opět požadovaným výsledkem, aby sloty `x` a `y` byly zařazeny do profilu. V druhém příkladu je pak žádoucí, aby slot `initTime` byl vložen do profilu. V těchto případech už musí být prozkoumána i těla metod, která objekt obsahuje ve svých slotech, aby šlo správně určit instanční sloty.

Současná implementace řeší problém takto: Překladač přidá do metody seznam se jmény slotů, které budou v objektu při provedení této metody vytvořeny. Pokud je pak takto označená metoda vložena do slotu v nějakém objektu, jsou jména slotů ze seznamu v metodě vloženy do jeho profilu. Toto řešení pokrývá oba výše zmíněné příklady.

V druhém případě existuje metoda, která vytváří slot `init`. Tento slot však nebude zařazen do profilu jako instanční, protože tato metoda je sice nad daným objektem vykonána, ale není vložena do slotu v objektu[26].

7.3.2 Chybná detekce slotů v profilu

Problém může nastat v případě, že jsou sloty objektů označených jako instance modifikovány jinak než výše popsané příklady. Například objekt přímo manipuluje se sloty jiného objektu. Obecně však považuji měnit sloty jiného objektu zvnějšku za špatný návrh programu. Je to stejné jako v jazyce s třídami měnit cizímu objektu přímo instanční proměnné, což například Smalltalk vůbec neumožňuje.

Výjimkou vnější modifikace objektu je definice prototypu. V tomto případě ale jsou sloty detekovány správně, protože metoda s touto operací není typicky vložena do slotů daného objektu.

Pokud ale nastane problém se špatným odhadem toho, které sloty mají být v profilu, tak to výsledek programu *nijak* neovlivní. Jediný možný důsledek je, že nemusí být plně využita inline cache a může dojít ke ztrátě výkonu při provádění programu.

[26] Zpráva `do:` pouze vykoná metodu nad objektem, který je příjemcem zprávy. Metoda ale není zapsána do žádného slotu.

Myslím ale, že rychlostní přínos inline cache je natolik velký a příčiny chybně vyplněného profilu zanedbatelné, že může být tato optimalizace standardní součástí VM. I v případě, že k problému dojde, je režie poměrně malá. Nechci však uživatele Kreatrix nutit, jak mají vypadat jejich programy. Proto, pokud by chtěli využít inline cache i v případech, se kterými tento detekční mechanizmus nepočítá, mají možnost. Součástí VM jsou metody, které umožňují uživateli vlastní zásah do profilu. Pokud by i přesto byla celá inline cache na překážku, může být vypnuta pomocí parametru `--disable-icache` v `configure`.

7.3.3 Změna p-typu při přidávání a aktualizaci slotů

Pro následující popis se omezím na případ, kdy má objekt jen jeden rodičovský slot, který navíc není měněn. Další případy jsou popsány v další podkapitole.

Když je prototyp naklonován, je vytvořen objekt – klon, který je prázdný. Nový objekt je označen jako instance, protože nemá vlastní sloty a tedy nemůže porušit pravidlo tím, že by obsahoval slot neuvedený v profilu. Objekty označené jako instance sdílejí stejný profil jako rodič, proto naklonovaný objekt získá referenci na profil z rodiče.

V případě, že je do objektu označeného jako instance vložen nový slot, mohou nastat dvě reakce.

Pokud se jméno nového slotu vyskytuje v profilu, pak objekt zůstane označený jako instance. Pokud je do instance zapsán slot, který není v profilu, je p-typ daného objektu změněn na prototyp. Již dále nemůže zůstat instancí, protože pro takto přidaný slot by mohl vracet jiný objekt, než obsahuje prototyp. Pro nový prototyp je vytvořena nová samostatná kopie profilu.

Pokud dojde k naklonování objektu, který není označen jako prototyp, dojde k jeho změně na prototyp. Tedy je změněn p-typ v objektu a je vytvořena kopie profilu. Toto je provedeno z důvodu, že některé další mechanizmy pracují s předpokladem, že objekt v rodičovském slotu každého objektu je vždy prototyp.

Pokud je do prototypu vložen nový slot, musí být ověřeno, jestli nedošlo k porušení konzistence inline cache. Detaily, jak konkrétně je cache opravována, jsou uvedeny v podkapitole 7.4.3.

Součástí profilu je také seznam prototypů, které mají za rodiče objekt, který vlastní daný profil. Dále v textu budu tento seznam prototypů označovat jako „seznam potomků profilu“. Předpokládám, že prototypů je vůči ostatním objektům poměrně málo a nevznikají a nezanikají příliš často, proto má udržování tohoto seznamu jen malou režii.

Pokud je pro daný objekt detekován nový instanční slot jednou z metod uvedených v předchozím textu, pak je do profilu vloženo nové jméno slotu. Pokud objekt nemá vlastní profil (tzn. je označen jako instance), tak je před touto operací změněn na prototyp, aby vlastní profil získal. Jméno slotu je pak rekurzivně vloženo také do všech dceřiných prototypů. Tím, že instance sdílejí profil svého prototypu, se změna projeví i ve všech profilech dceřiných objektů.

7.3.4 Aktualizace profilu při změně rodičovského slotu

Předchozí případy popisovaly situaci, kdy má každý objekt právě jeden rodičovský slot, který je nastaven při naklonování a dále není měněn. Ačkoliv většina objektů se takto chová, v Kreatrix mohou být rodičovské sloty za chodu měněny. Každý objekt také může mít libovolný počet těchto slotů včetně případu, kdy nemá žádný.

Současná implementace detekce prototypů a instancí kvůli zachování jednoduchosti neřeší problém více rodičovských slotů v plném rozsahu. Detekce využívá pouze první rodičovský slot.

Problémem tohoto přístupu je nemožnost v inline cachi uchovávat sloty z dalších rodičů. Tento problém se však týká pouze objektů označených jako instance a mající více rodičů. Takové objekty jsou však velmi vzácné. Výhodou tohoto omezenějšího systému je mnohem jednodušší návrh a rychlejší zpracování v případě jednoho rodičovského slotu, což je drtivá většina případů.

V případě, že dochází ke změně rodičovského slotu, mohou nastat dva případy, podle toho, jaký p-typ objekt má.

Pokud je měněn rodičovský slot objektu typu instance, je postupováno takto: Pokud je změněn rodičovský slot na první pozici, tak je změněn profil podle nového rodiče. Navíc se pak musí ještě ověřit, zda-li objekt stále může zůstat instancí v rámci nového profilu. Ověří se, že všechny sloty, které obsahuje, jsou také zahrnuty v novém profilu. Pokud podmínka není splněna, je typ objektu změněn na prototyp. V případě změny rodičů na jiné než první pozici není třeba dělat nic dalšího.

V případě, že p-typ je prototyp, pak musí být vymazána inline cache tykající se daného prototypu a všech prototypů, které jsou jeho potomky. Pokud je přidáván nový rodičovský slot na první pozici, jsou do profilu z profilu rodiče zkopírovány jména slotů. Zkopírování jmen slotů se provede také rekurzivně do všech objektů v seznamu potomků profilu. Při změně rodiče na jakékoli pozici musí být také patřičně aktualizován seznam potomků profilu.

Po změnách rodičovského slotu nejsou z profilu nikdy odebírány sloty, i když by podle nových rodičů už v profilu nemusely být. Důvodem je možnost, že daný prototyp již může mít potomky, které nějaký slot z profilu mají. Z důvodů sdílení profilu mezi prototypem a instancemi by odebráním jmen slotů z profilu mohl vzniknout chybný stav, ve kterém by tyto objekty již měly neplatné označení vzhledem k profilu.

Aby však došlo k opravě, musela by současná implementace projít všechny existující objekty v systému a provést nápravu. Vzhledem k tomu, že se jedná o zcela okrajové chování a režie může být značná, současná implementace odstraňování jmen slotů z profilu neprovádí. Z hlediska korektnosti provádění programu však k žádnému problému nedochází.

7.4 Implementace inline cache v Kreatrix

V momentě, kdy správně funguje systém detekce prototypů a instancí, tak se implementace inline cache principiálně neliší od jazyka s třídami. Jen místo testování ukazatele instance na třídu je testován profil.

7.4.1 Aktivace inline cache

V Kreatrix může být inline cache použita pro jakýkoliv blok kódu (metoda, closure). K aktivaci může dojít manuálně zasláním zprávy `insertInlineCache`. Může být také zapnuta automaticky, pokud je spuštěna podpora hledání kritických míst v programu, viz kapitola 7.5.

Při zapnutí inline cache dojde k následujícím dvěma krokům: nahrazení instrukcí a alokaci inline cache. V rámci bytekódu pro daný blok kódu jsou některé instrukce nahrazeny variantami, které používají inline cache. Samotná inline cache je pak tabulka, ve které počet řádků odpovídá počtu nahrazených instrukcí v kódu.

Nové instrukce jsou tyto:

- `UNARY_MSG_C`
- `BINARY_MSG_C`
- `KEYWORD_MSG_C`
- `LOCAL_UNARY_MSG_C`
- `LOCAL_KEYWORD_MSG_C`

Tyto instrukce nahrazují instrukce, které se jmenují stejně, jen bez přípony „`_C`“. Počet parametrů zůstává stejný jako u původních instrukcí. První parametr však u původních instrukcí vyjadřuje pozici v tabulce symbolů a tento symbol slouží jako jméno zprávy. U nových instrukcí má první parametr význam indexu do tabulky inline cache. Jméno zprávy pro danou instrukci je pak součástí inline cache.

To, že nové instrukce mají stejnou délku jako instrukce, za které jsou nahrazovány při aktivaci inline cache, má několik výhod. Přináší to snadnější nahrazení instrukcí tam i zpět. Takže místa, u kterých se detekuje chování pro inline cache nevhodné, mohou být velmi rychle změněna zpět na původní instrukci.

Další výhodou zachování původní délky instrukcí je eliminace problému se změnou instrukcí u metod, které mají aktivační záznam na zásobníku (jsou tedy při nahrazování uprostřed vykonávání). Ukazatel na aktuální instrukci nemusí být měněn a při dalším provedení instrukce může být rovnou vykonáván změněný kód.

Poslední výhodou je pak bezproblémové spojení s rozšířenými instrukcemi (kapitola 8.2). Tyto instrukce oproti základním instrukcím obsahují skoky v bytekódu. Kdyby se měnily pozice instrukcí, muselo by dojít k přepočítání těchto skoků na správné místo.

7.4.2 Provádění kódu s inline cache

Inline cache v Kreatrix je tabulka, která je přiřazena bloku kódu. Každý řádek je určen pro jednu instrukci v kódu, která využívá cache. V každém řádku jsou uloženy následující údaje: profil, vlastník slotu, cachovaný objekt a jméno zprávy. Po inicializaci inline cache jsou všechny položky v tabulce kromě jména zprávy nastaveny na `NULL`. Jméno zprávy je vyplněno podle původního jména zprávy v instrukci.

Údaj o profilu slouží k ověření typu objektu, jestli může být cache použita. Položka „vlastník slotu“ obsahuje objekt, ve kterém se nachází slot, který je nalezen při zasílání zprávy[27]. Tato informace je zde z důvodu, aby mohlo použití cache plnohodnotně nahradit vyhledávání ve slotech[28].

Když zpracování metody dorazí k instrukci využívající cache, tak místo zaslání zprávy nejprve dojde k ověření cache. Kontroluje se, jestli objekt, kterému má být zpráva zaslána, má stejný profil, jaký obsahuje inline cache. Pokud je tato podmínka splněna, tak je použit objekt v inline cache. Dále následuje stejná aktivace objektu jako v „normálním“ případě zaslání zprávy, ale bez nutnosti vyhledávat slot.

Může však nastat případ, ve kterém nejde cache použít. Z důvodu nesplnění podmínek pro použití nebo protože je prázdná. V takovém případě zkusí virtuální stroj aktualizovat cache podle aktuálního objektu, který daným místem prochází. Pokud jméno zasílané zprávy není v profilu objektu, je provedeno vyhledání slotu. V případě, že je vyhledání úspěšné, je výsledek zapsán do cache, ukazatel na aktuální instrukci je posunut zpět a celá instrukce je provedena znovu.

V případě, že se nepodaří vyplnit inline cache, je aktuální instrukce změněna zpět do původní varianty bez využití cache. Když je v daném místě cílem zprávy objekt, pro který z nějakého důvodu není možné vytvořit cache, tak je pravděpodobné, že se tak stane zase. Proto je v tomto místě cache zrušena, než aby byl riskován případ, že dojde ke zbytečné režii spojené s opakovaným neúspěšným pokusem vyplnit cache.

Současná implementace používá tuto defenzivní strategii, protože sběr dalších informací pro získání celkového přehledu o dané instrukci by znamenal velkou režii. Navíc v drtivé většině míst je zrušení inline cache tím lepším postupem. V nejhorším případě dojde ke zpomalení na původní úroveň bez inline cache.

7.4.3 Údržba inline cache v případě změny prototypů

V případě, že dojde k nějaké změně v prototypu, může dojít k tomu, že v cache jsou chybné údaje. Proto musí po takových změnách dojít k pročištění tabulek pro inline cache. Jsou vymazána všechna místa ve všech tabulkách, která se týkají změněného prototypu a jména slotu. Pokud je měněn rodičovský slot, pak dojde k vymazání všech záznamů týkajících se daného prototypu, bez ohledu na jméno zasílané zprávy. Vymazání je pak provedeno rekurzivně pro všechny prototypy v seznamu potomků profilu.

Všechny bloky kódu, které obsahují inline cache, jsou uspořádány do spojitého seznamu. Tento seznam je pak použit při opravě cache, aby nemusely být prohledávány všechny objekty a v nich hledány ty objekty, které obsahují inline cache.

Různé situace, kdy musí dojít k opravě inline cache, jsou uvedeny v regresivním testu v souboru: `kreatrix/tests/test_inlinecache.kx`.

[27] Informace o vlastníku zprávy je použita při „přeposílání“ zprávy. Bližší detaily jsou uvedeny v dokumentaci ke klíčovému slovu `resend` (`http://www.kreatrix.org/manual-messages`).

[28] Vyhledání ve slotech ve skutečnosti poskytuje navíc ještě informaci o příznacích slotu. Do cache jsou však ukládány jen objekty ze slotů, které nemají nastaveny žádné příznaky.

7.5 Hledání výkonnostně kritických míst v programu

Využití inline cache má význam především u metod, které jsou vykonávány často. Nemá smysl vkládat inline cache do každé metody, protože by docházelo ke zbytečné spotřebě prostředků, které se nijak nemusí projevit.

Z toho důvodu jsem do virtuálního stroje přidal také podporu pro detekci výkonnostně kritických míst v programu. Jsou to místa, kde program stráví většinu času a vyplatí se tedy maximálně optimalizovat. Současná implementace hledá kritická místa tak, že sleduje počet spuštění metod. Pokud překročí tento počet určitou hranici, dojde k vložení inline cache do dané metody.

Aby však nedocházelo ke zbytečné režii spojené s počítáním spuštění i poté, co dojde k optimalizaci, je tato detekce realizována pomocí instrukce `HOTSPOT_PROBE`. Tato instrukce je vložena při načítání bytekódu do každé metody hned na první místo. Při jejím zpracování dojde ke zvýšení čítače spuštění a dojde k ověření, jestli už nebylo dosaženo hranice pro optimalizaci. Pokud hranice dosaženo bylo, dojde k vložení inline cache a instrukce `HOTSPOT_PROBE` je odstraněna, takže k další inkrementaci počítadla nedochází.

Toto řešení je oproti implementaci, kdy dochází k inkrementování počítadla a ověření počtu při každém spuštění metody i po detekci kritického místa, ve výkonnostních testech asi o 2–4% rychlejší.

Výše popsané řešení naráží na jeden problém při použití rozšířených instrukcí (viz kapitola 8.2). Při této optimalizaci jsou některé vnořené bloky rozepsány přímo do těla metody. K vykonání hotspot instrukce však dochází pouze na začátku provádění bloku kódu. To může způsobit v případě cyklů pozdější detekci často vykonávaného místa, než v případě běhu bez rozšířených instrukcí.

U déle běžícího programu je problém v podstatě eliminován. Zkreslení ale může nastat například při jednorázových testech výkonu. V nejhorším případě lze celou situaci řešit manuálním zapnutím inline cache pro danou metodu.

7.6 Výsledky testů při použití inline cache

V tabulce 6 je uvedeno měření výkonnostních testů verze 0.12.0 s deaktivovanou a aktivovanou podporou inline cache. Deaktivovaná podoba byla zkompilována s příznaky `--disable-icache` a `--disable-hotspot`. V obou měřeních je aktivována také optimalizace pomocí rozšířených instrukcí.

7.7 Polymorfní inline cache

V předchozím textu byla vždy inline cache popisována v základní variantě, kdy jde pro každé odeslání zprávy držet jen jednu položku v cache. Pokud je zpráva zaslána jinému typu objektu, je cache aktualizována a starý obsah je zahozen. Problém může nastat v případech, kdy daným místem prochází pravidelně nějaká množina objektů různých typů a navzájem si tyto průchody ničí záznam v cache.

Řešením tohoto problému je použití polymorfní inline cache. Poprvé byl tento princip implementován v jazyce SELF[3]. Inline cache v této podobě již nemá jen jeden zá-

Benchmark	Čas 1 (ms)	Čas 2 (ms)	Rozdíl	Rozdíl v %
bigobject	608	368	-240	-39,473%
gc	2205	2753	548	24,852%
mdispatch	773	626	-147	-19,016%
t1_recursive	448	305	-143	-31,919%
t1_sort	1009	809	-200	-19,821%
t1_sum	132	141	9	6,818%
ccounter	57	40	-17	-29,824%
graph	106	81	-25	-23,584%
wcounter	438	304	-134	-30,593%
world	1848	1329	-519	-28,084%
sgfplayer	514	390	-124	-24,124%

Tabulka 6: Výsledky výkonnostních testů s aktivovanou inline cache.

znam pro dané místo, ale záznam samotný má podobu tabulky. V této tabulce pak může být uloženo více záznamů a pokud prochází daným místem více typů objektů, všechny jsou uloženy do tabulky a nedochází k neustálému přepisování cache.

Při použití cache se pak sekvenčně prochází tabulka a hledá se záznam, který souhlasí s typem objektu. Pokud však daným místem prochází příliš mnoho typů objektů, mohla by režie příliš narůst. Proto, pokud množství záznamů v tabulce přesáhne nějaký limit, je dané místo označeno jako „megamorphic“ a celá inline cache je v daném místě odstraněna.

V současné implementaci Kreatrix není polymorfní inline cache zahrnuta. Principiálně by však v rozšíření současné implementace neměl být žádný zásadní problém.

7.8 Další metoda detekce instancí a prototypů

Způsob získávání informací pro rozlišení prototypů a instancí uvedený v této podkapitole nemá s postupem, který byl v předchozím textu popsán, nic společného. Chronologicky je tato alternativní metoda předchůdcem metody výše popsané. Tento postup měl za cíl nejenom detekci prototypů a instancí, ale měl také řešit rychlé vyhledávání ve slotech a navíc na některých místech ušetřit paměť. Nakonec ale tato metoda, pro velkou komplikovanost a ne příliš výrazné výkonnostní zlepšení, nebyla zahrnuta do Kreatrix. Uvádím ji zde však jako další možnost přístupu k problému. Vývojovou větev s implementací je možné nalézt na přiloženém CD, jedná se o větev `kreatrix-sc`.

Struktura `KxObject` je rozšířena o ukazatel na profil. Profil má však jinou funkci než ve výše popsaném postupu. V případě, že objekt je prototyp, jsou veškeré jeho sloty uloženy v hashovací tabulce[29], která je součástí profilu. Prototyp tedy nedrží sloty přímo, ale prostřednictvím profilu.

Součástí profilu jsou také všechny sloty předků, které jsou dosažitelné zasláním zprávy majiteli profilu. Z toho důvodu již není nutné prohledávání do hloubky přes rodičovské

[29] V implementaci byl použit Cuckoo hashing[6]

sloty v případě zaslání zprávy. Stačí jedno vyhledání v tabulce slotů v profilu a okamžitě je jasné, jak bude daný objekt reagovat.

Udržovat však takový profil pro každý existující objekt by bylo velmi paměťově náročné, například sloty ze základního objektu by byly obsaženy v téměř všech profilech. Proto by bylo žádoucí, aby bylo možné profily mezi objekty sdílet. Zde je využita myšlenka detekování prototypů a instancí, i když v trochu jiné podobě.

Prázdný objekt může bez obav používat profil svého rodiče, protože reaguje na vyhledávání ve slotech stejně jako rodič. Pokud je do naklonovaného objektu vložen slot, je třeba změnit profil. V případě změny instančního slotu se jménem X je vytvořen nový profil, který je kopií toho původního, jen u jména X má místo reference na objekt nastavenu hodnotu `NULL`. `NULL` při vyhledávání slotů říká, že objekt v profilu není a je jej nutné vyhledat v tabulce slotů pro konkrétní objekt.

Protože v případě změny instančního slotu je v profilu `NULL` a nikoliv konkrétní hodnota, tak mohou objekty označené jako instance, které mají stejného rodiče a stejná jména změněných instančních slotů, profil sdílet. Proto je v profilu u každého slotu přidána položka, na jaký profil se má přejít, pokud je daný slot změněn. Pokud je vytvořen nový profil z důvodu změny instančního slotu, tak je do tohoto místa zapsán a další objekty už pak takový profil nevytváří, ale využívají existující.

Teoreticky však může být pro objekt s n instančnímy sloty $n!$ možností, jakou posloupností změn mohou být sloty vyplňovány. Proto může také vzniknout $n!$ profilů pro jeden prototyp. Ve většině případů jsou však instanční sloty vyplňovány jen několika způsoby. Ve velkém množství případů pak pouze jedním způsobem a vzniká tedy jen n profilů pro každý prototyp (bez ohledu na to kolik existuje naklonovaných instancí).

Pokud vyhledávání v profilu najde ve slotu místo objektu `NULL`, musí být obsah slotu nalezen v samotném objektu. Protože však profil zároveň jednoznačně určuje, v jakém pořadí byly sloty do objektu přidávány, je v profilu také poznačen index do tabulky slotů konkrétního objektu, kde se hledaný objekt nachází. Takže slot je přímo nalezen bez nutnosti procházet celou tabulku slotů. Díky tomu ani tato vnitřní tabulka slotů nemusí obsahovat jména slotů, protože ty jsou součástí profilu.

V této metodě je také identifikace instančních slotů řešena jinak, než postup popsaný výše. V této implementaci je u neinstančních slotů počítáno, kolik potomků daného objektu si určitý slot vytvořilo. Pokud dojde k překročení limitu, je daný slot označen jako instanční. Tento postup nemá některé nevýhody, které má postup popsaný na začátku této kapitoly, přináší však jiné problémy.

Hlavní problém pramení z toho, že informace o tom, jaké sloty jsou instanční, přichází se zpožděním v době, kdy už existují potomci, kteří byli označeni jako prototypy. Proto v případě označení slotu jako instančního musí být u potomků objektu přehodnoceno, zda-li nemohou být označeni jako instance.

Navíc může být problematické najít správný limit, kdy označit slot za instanční. U objektu, ze kterého je naklonováno hodně prototypů, například základní objekt `Object`, mohou být tímto způsobem chybně označeny některé sloty jako instanční. Opačný problém pak může potkat prototyp, který má počet potomků pod limitem.

8 Optimalizace bytekódu

V následující kapitole budou rozebrány změny, které se dotýkají převážně bytekódu. Změny v implementaci se oproti předchozím kapitolám týkají nejen interpretru, ale především překladače. Změny se ale týkají jen zadní části překladače pracující s bytekódem. Syntaxe jazyka zůstala nezměněna.

8.1 Nové instrukce v bytekódu

Vznik nových a úprava stávajících instrukcí je především důsledek celkové změny pohledu na instrukce tak, aby se zjednodušilo rozhraní pro práci s nimi. Z toho pak těží nejen samotný překladač, ale také modul pro dekompilaci a následně inline cache (viz kapitola 7) a JIT překladač (viz kapitola 9).

8.1.1 Nový formát instrukcí

Předchozí formát instrukcí byl poměrně volný. První byte obsahoval identifikační číslo instrukce a další data byla dekódována na základě typu instrukce a mohla mít libovolný formát. Například za prvním bytem instrukce pro vložení řetězce na zásobník následuje samotný obsah řetězce. Instrukce tedy mohly být různě dlouhé.

Nové instrukce mají pevný formát. Za prvním bytem, který identifikuje instrukci, může následovat nula až n bytů náležících k instrukci. Počet bytů je pro každou instrukci pevně dán. Každý byte instrukce je pak vždy chápán jako samostatné číslo, nejčastěji jako index do nějaké pomocné tabulky.

Kompletní seznam instrukcí a počty jejich parametrů je možné najít v souboru `kreatrix/src/compiler/kxinstr.c`.

8.1.2 Přístup k lokálním slotům a aktivačnímu záznamu

Původní implementace jazyka Kreatrix se snaží používat základní infrastrukturu objektů, slotů a zpráv i v případě aktivačního záznamu. Aktivační záznam je tedy také reprezentován objektem a lokální sloty a parametry jsou normální sloty v aktivačním záznamu. Při spouštění closures pak aktivační záznamy byly propojeny rodičovským slotem. Proto šlo zprávou přistupovat k lokálním slotům aktivačního záznamu, v jehož kontextu closure běží.

Tento přístup má řadu výhod, dobrou flexibilitu, využívá stávající kód, k introspekci aktivačních záznamů je přistupováno stejně jako k introspekci ostatních objektů. Nevýhodou je kromě složitější sémantiky některých speciálních metod především velmi vysoká režie. Režie vzniká především při vytváření objektu pro každý aktivační záznam a ten se vytváří při každém spuštění metody. Další režie vzniká při přístupu k lokálním slotům, kdy dochází k normálnímu prohledávání slotů. Všechny tyto akce jsou vykonávány velmi často, proto se každé zdržení značně projeví.

Úpravy, které jsou popsány v následujícím textu, mírně mění sémantiku aktivačních záznamů, porušuje to tedy jedno z mých návrhových pravidel. Výsledná sémantika je

však přehlednější a odstraňuje některé speciální případy. Navíc ve výsledku se jedná jen o změnu v chování v poměrně okrajových případech a dopad na existující kód je minimální[30]. Výkonnostně se jedná o poměrně velký přínos a je možné na tuto úpravu navázat další optimalizace. Proto jsem se rozhodl provést tuto změnu.

Podstatou změny je, že aktivační záznam již není reprezentován objektem. Instrukce pro odeslání lokálních zpráv, které mají stejné jméno jako lokální sloty nebo parametry, jsou nahrazeny speciálními instrukcemi. Lokální sloty jsou uloženy v poli. V tomto poli leží na začátku parametry a za nimi pak lokální sloty definované v metodě.

Nové instrukce:

- `PUSH_LOCAL` – Vložení obsahu lokálního slotu na zásobník.
- `PUSH_OUTER_LOCAL` – Vložení obsahu lokálního slotu z vnějšího aktivačního záznamu na zásobník.
- `UPDATE_LOCAL` – Odebere ze zásobníku objekt a uloží ho do lokálního slotu.
- `UPDATE_OUTER_LOCAL` – Odebere ze zásobníku objekt a uloží ho do vnějšího lokálního slotu
- `PUSH_SELF` – Vloží na zásobník `self`.
- `PUSH_LOCAL_CONTEXT` –Vloží na zásobník `LocalContext`.

Instrukce `PUSH_LOCAL` a `UPDATE_LOCAL` mají jeden parametr a tím je pořadové číslo lokálního slotu. Instrukce `PUSH_OUTER_LOCAL` a `UPDATE_OUTER_LOCAL` jsou použity při čtení a zápisu lokálních slotů do vnějších aktivačních záznamů při použití closures. Tyto instrukce mají dva parametry, pořadové číslo lokálního slotu a počet zanoření vůči aktivačnímu záznamu, který drží lokální sloty.

Instrukce `PUSH_SELF` je použita, pokud je zaslána lokální zpráva `self`. V předchozích verzích byl do objektu aktivačního záznamu vložen kromě parametrů a lokálních slotů také slot `self`[31]. Tato instrukce tedy slouží jako náhrada za tento slot.

Instrukce `PUSH_LOCAL_CONTEXT` je vložena při zaslání lokální zprávy `localContext`. Tato instrukce nijak nekoresponduje s chováním předchozích verzí, ale je zde kvůli zachování možnosti introspekce aktivačního záznamu. Aktivační záznam není objekt a tedy původní možnosti introspekce již nefungují. Tato instrukce umožňuje vytvořit objekt `LocalContext`, který zpřístupní data v aktivačním záznamu.

V tabulce 7 je uvedeno výkonnostní srovnání starých a nových instrukcí.

[30] Jediný případ, kdy musel být upraven zdrojový kód napsaný v jazyce Kreatrix, byl jeden regresivní test, který kontroloval okrajový případ. Žádný další kód v důsledku této sémantické změny nemusel být změněn.

[31] Tento slot obsahoval objekt, který byl příjemcem zprávy, při které došlo ke spuštění metody. Jedná se tedy o stejný význam, jaký mají klíčová slova `self` nebo `this` např. ve Smalltalku nebo Javě.

Benchmark	Čas 1 (ms)	Čas 2 (ms)	Rozdíl	Rozdíl v %
gc	2166	2168	2	0,092%
mdispatch	1946	1617	-329	-16,906%
t1_recursive	1324	1035	-289	-21,827%
t1_sort	2216	1268	-948	-42,779%
t1_sum	408	220	-188	-46,078%
ccounter	216	157	-59	-27,314%
graph	242	177	-65	-26,859%
wcounter	669	465	-204	-30,493%
world	4825	3301	-1524	-31,585%
sgfplayer	1125	818	-307	-27,288%

Tabulka 7: Výsledky výkonnostních testů při použití starého formátu instrukcí (Čas 1) a nového formátu instrukcí (Čas 2).

8.1.3 Cache literálů

Spolu s novými instrukcemi bylo také změněno chování instrukcí, které vkládají na zásobník literály. Předchozí podoba instrukcí obsahovala za prvním bytem data, ze kterých byl pak vytvořen daný objekt. Pro každý typ literálu existovala zvláštní instrukce.

V nové podobě instrukcí už existují jen dvě instrukce: `PUSH_LITERAL` a `PUSH_LIST`[32]. Parametrem instrukce `PUSH_LITERAL` je pozice v tabulce literálů. Samotná tabulka literálů je vytvořena při natažení metody do virtuálního stroje. Literály jsou tedy vytvořeny již před prvním spuštěním. Před touto optimalizací byly vytvářeny znovu vždy v každém spuštění metody.

8.2 Rozšířené instrukce

Základní bytekód je sám o sobě velmi abstraktní, kromě několika pomocných instrukcí umí „jen“ zasílat zprávy a vkládat literály. Bytekód nijak nedefinuje konstrukce pro podmínky, cykly a další základní funkcionalitu. Toto vše je zahrnuto v základní knihovně. Z toho plyne velká flexibilita a je možné dynamicky téměř cokoli změnit.

Nevýhodou takto abstraktního bytekódu je praktická nemožnost provádět optimalizace v době překladu. Překladač v době kompilace nemá informace, jak budou dané objekty reagovat na zaslané zprávy.

Jedním z možných řešení je upustit od úplné abstrakce všech zpráv a některým jménům často používaných zpráv přiřadit konkrétní chování. Například zprávy `ifTrue:` a `ifFalse:` jsou nejčastěji použity v souvislosti podmíněného vykonání kódu. Pokud by chování těchto zpráv již bylo implicitně definováno, tak může překladač tuto informaci využít a nahradit dané zaslání zprávy instrukcemi pro podmíněný skok. Tím se

[32] Tato instrukce vytvářející seznam zůstala zachována, protože seznam není „normální“ literál. Jeho obsah je jednak vytvářen dynamicky a navíc není neměnný, což neumožňuje cachování.

ušetří poměrně velká režie spočívající ve vytváření closure, vyhledání slotu a následném spuštění a provedení metody.

Značnou nevýhodou tohoto řešení je zhoršení ortogonality, protože vzniká množina zpráv, které mají speciální pravidla chování. Navíc ve výše popsaném příkladu nejen že nejde redefinovat reakce na tyto zprávy pro standardní objekty, ale nelze tato jména zpráv nadále použít i pro potřeby vlastních objektů[33].

Původně jsem se touto optimalizací z důvodu těchto nevýhod nechtěl hlouběji zabývat. Po provedení několika experimentálních testů se však ukázalo, že výkonnostní zlepšení může být velmi znatelné.

Do bytekódu byly proto zahrnuty rozšiřující instrukce pro některé základní konstrukce. Aby však došlo k minimalizaci ztráty flexibility z výše popsaných důvodů, mají rozšiřující instrukce trochu složitější chování. Tyto instrukce jsou schopné, při nesplnění vstupních podmínek, zaslat normální zprávu. Tedy jako by v daném místě žádná zvláštní instrukce nebyla. Jména zpráv tedy nejsou nijak speciální a uživatel je může volně používat ve svých objektech bez omezení.

Navíc je možné tuto vlastnost jednoduše vypnout, protože se jedná pouze o nastavení překladače. Pokud chce uživatel plnou flexibilitu, stačí, aby překladač tyto instrukce do výsledného bytekódu nevkládal. Vykonávání takto neoptimalizovaného kódu tedy není nijak penalizováno, pokud je ve VM zakompilována podpora pro rozšířené instrukce.

8.2.1 Obecné fungování rozšířených instrukcí

V jazyce Kreatrix jsou konstrukce pro řízení toku kódu postaveny na closures. Pomocí closures a metod ve standardní knihovně jsou řešeny podmíněné skoky a cykly. Následující optimalizace funguje tak, že obsah closure je zkopírován přímo do bytecodu rodičovské metody. Řízení toku kódu je pak realizováno pomocí rozšířených instrukcí.

Aby mohla být tato optimalizace provedena, musí být v daném místě zasílána zpráva s konkrétním jménem. Dále, pokud je očekávána jako parametr nebo jako cíl zprávy closure, musí být v daném místě vkládána jako literál, aby překladač měl k dispozici konkrétní kód pro rozepsání. Samotná instrukce, vkládající closure na zásobník (`PUSH_BLOCK`), je pak odstraněna. Rozšířené instrukce, které by daný blok potřebovaly využívat v nerozepsané podobě, si nesou pořadové číslo bloku ve svých parametrech.

Rozepsání closure do rodičovského bloku je prováděno následovně:

1. Vložení literálů, lokálních slotů a seznamu s čísly řádků[34].
2. Vytvoření nepřímých bloků (viz dále).

[33] Například v implementaci Smalltalku Squeak (`http://www.squeak.org/`) není možné používat zprávy `ifTrue:` a `ifFalse:` pro vlastní objekty. Naopak v implementaci GNU Smalltalk (`http://smalltalk.gnu.org`) to možné je.

[34] Jedná se o tabulku, ve které jsou čísla řádků ze zdrojového souboru, na kterých se nachází příkazy pro odeslání zprávy. Tato tabulka je použita při vyhození výjimky. Pomocí ní může být zjištěno, kterými místy ve zdrojovém souboru procházel tok programu, když došlo k výjimce.

3. Nahrazení instrukce pro zaslání zprávy rozšířenou instrukcí. V případě některých instrukcí je provedeno vložení dalších pomocných instrukcí. Například v případě cyklů je vložena další instrukce za místo, do kterého je rozepsán bytekód z closure.

4. Samotné rozepsání kódu. Některé instrukce musí být při rozepsání upraveny. Například pokud instrukce odkazuje do tabulky literálů nebo lokálních slotů, musí být změněny indexy.

Samotná closure po rozepsání do rodičovské metody však stále existuje, aby mohl být případně vykonán kód v neoptimalizované podobě. Nevýhodou tohoto řešení je vyšší paměťová náročnost, protože bytekód, který dělá ve výsledku totéž, je přítomen dvakrát. Tato nevýhoda je vyvážena menší ztrátou flexibility, která by jinak rozšířené instrukce postihovala.

Pro snížení duplicit byl bytekód rozšířen například o nepřímé bloky. Díky nim může `KxCodeBlock` získat reference na jiné instance `KxCodeBlock`, které nejsou definovány přímo uvnitř něj. Pomocí tohoto mechanizmu může překladač zabránit duplicitním blokům, které by jinak vznikaly po rozepsání closure, která obsahuje vlastní closure.

8.2.2 Instrukce pro podmíněný skok

V této části budou popsány rozšířené instrukce pro podmíněné vykonání kódu. Základní knihovna poskytuje v objektech true a false tyto zprávy:

- `ifTrue:`
- `ifFalse:`
- `ifTrue:ifFalse:`
- `ifFalse:ifTrue:`

Všechny metody očekávají jako parametr(y) bloky kódu bez parametrů. Objekt true reaguje na zprávu `ifTrue:` tak, že parametr (blok) vyhodnotí a na zprávu `ifFalse:` neprovede nic. Přesně opačně pak reaguje objekt false. Zprávy `ifTrue:ifFalse:` a `ifFalse:ifTrue:` pak fungují analogicky a slouží ke stejnému účelu jako konstrukce „if/else“ např. v C nebo Javě.

Při optimalizaci pomocí rozšířených instrukcí jsou tyto zprávy nahrazeny těmito instrukcemi: `IFTRUE`, `IFFALSE`, `IFTRUE_IFFALSE` a `IFFALSE_IFTRUE`. První dvě instrukce přijímají dva parametry, číslo bloku a pozici pro skok za rozepsaným bytekódem z bloku.

Při provádění těchto instrukcí je nejprve ověřeno, zda na vrcholu zásobníku leží objekt true nebo false. Pokud by měl být parametr vykonán, pak je dále prováděn bytekód za instrukcí. Protože za rozšířenou instrukcí leží vložený kód z bloku, je obsah bloku vykonán, aniž by musel být vytvářen objekt ScopedBlock a bez celé režie spojené s aktivací CodeBlocku. Pokud daný blok nemá být vykonán, je proveden skok v bytekódu podle druhého parametru, tedy za vložený bytekód.

Pokud objekt na vrcholu zásobníku není true nebo false, pak je danému objektu zaslána původní zpráva (`ifTrue:` nebo `ifFalse:`, podle typu rozšířené instrukce). Parametrem zprávy je blok získaný z prvního parametru rozšířené instrukce. Následně je pak proveden skok za kód, který je vložen z bloku. Tímto dojde ke stejnému efektu, jako kdyby byla vykonána pouze instrukce pro zaslání zprávy.

Instrukce `ifTrue:ifFalse:` a `ifFalse:ifTrue:` mají tři parametry: číslo bloku, pozici pro skok mezi bloky (v těchto instrukcích jsou rozepsány dva bloky) a pozici pro skok za oba bloky. Celá funkčnost je analogická s předchozím popisem. Jen za první rozepsaný blok je navíc vložena instrukce `JUMP` pro nepodmíněný skok, po kterém je skočeno za druhý rozepsaný blok.

8.2.3 Ukázka bytekódu při použití rozšiřujících instrukcí pro podmíněné vykonání kódu

Zdrojový kód v jazyce Kreatrix:

```
{
    (x < 2) ifTrue: [ "X je menší než 2" println ].
}
```

Bytekód bez optimalizací (hlavní blok metody, closure má vlastní blok kódu):

```
send_local_unary 0 (#x)
push_literal 0 (2)
send_binary 1 (#<)
push_block 0
send_keyword 2 1 (#ifTrue:)
pop
return_self
```

Bytekód s optimalizací pomocí rozšířených instrukcí. Obsah closure je vložen přímo do těla metody:

```
send_local_unary 0 (#x)
push_literal 0 (2)
send_binary 1 (#<)
iftrue 0 4
push_literal 1 ("X je menší než 2")
send_unary 3 (#println)
pop
return_self
```

8.2.4 Instrukce pro cyklus – Předem známý počet opakování

Další oblastí, pro kterou byly začleněny rozšířené instrukce, jsou cykly. Tato oblast je rozdělena na tři části, protože každá ze tří možností, jak vytvořit cyklus, funguje jinak

než ostatní. Základní myšlenka při překladu a provádění bytekódu však zůstává stejná jako u podmíněného skoku.

Tato část se zabývá cyklem, u kterého je znám počet opakování před započetím cyklu. Tento cyklus poskytuje standardní knihovna pomocí metod v objektu Integer. Tyto metody jsou ve slotech: `repeat:`, `to:do:` a `to:by:do:`.

- `repeat:` – Parametrem je blok bez argumentů. Tento blok je pak opakovaně vykonáván v závislosti na hodnotě příjemce zprávy.
  Například: `10 repeat: [ doSomething ]`
- `to:by:do:` – První dva parametry musí být celé číslo. Jako třetí parametr je očekáván blok s jedním argumentem. Příjemce zprávy je startovní hodnota, se kterou je blok vyhodnocen. Pak je k hodnotě postupně přičítán druhý parametr, dokud není dosaženo prvního parametru. Po každém přičtení je blok vyhodnocen s aktuální hodnotou.
  Například: `0 to: x by: 2 do: [ :i | i println ]`.
- `to:do:` – Stejné chování jako `to:do:by:` jen přičítaná hodnota je 1.

Tyto zprávy jsou pak nahrazeny instrukcemi `REPEAT`, `TODO` a `TOBYDO`. Parametry instrukcí jsou: pozice pro skok za koncovou instrukci (viz dále) a číslo bloku. U druhé a třetí instrukce je navíc číslo lokálního slotu. První dva parametry slouží podobně jako u předchozích instrukcí k zaslání původní zprávy v případě, že nejsou splněny podmínky pro spuštění optimalizace. Podmínky pro tyto instrukce jsou: příjemce zprávy musí být celé číslo a případné parametry mimo posledního musí být také celá čísla.

Za rozepsaný blok jsou pak vkládány instrukce `REPEAT_END`, `TODO_END` nebo `TOBYDO_END`. Tyto instrukce zajistí kontrolu čítače a případný skok kvůli opakování cyklu.

Když vykonávání dorazí do koncové instrukce, je na zásobníku objekt, který vznikl při vykonání kódu, který byl získán rozepsáním bloku. Tedy jako by byl blok vykonán. Pod tímto objektem leží na zásobníku čítač, koncová hodnota čítače (v případě `TODO` a `TOBYDO`) a změna čítače (v případě `TOBYDO`).

Koncová instrukce ověří pomocí čítače, jestli cyklus končí. Pokud končí, pak jsou tyto pomocné objekty odstraněny a následuje vykonávání kódu za koncovou instrukcí. Pokud cyklus bude opakován, je odstraněn objekt na vrcholu zásobníku (objekt vzniklý z předešlého výpočtu). Následně je proveden skok na místo, kde začíná kód pro cyklus.

8.2.5 Instrukce pro cyklus – Podmíněné opakování

Součástí standardní knihovny jsou metody, které slouží k opakovanému vykonávání kódu, dokud je splněna nějaká podmínka. Jedná se o metody objektu ScopedBlock: `whileTrue`, `whileFalse`, `whileTrue:` a `whileFalse:`.

Ukázka použití:

```
/* Ulož do slotu 'char' první znak ze standardního vstupu,
   který není mezera */
[ char << stdin readCharacter.  char == ' ' ] whileTrue.

// Spočítá počet řádků v souboru
count = 0.
[ file readLine isNil  ] whileFalse: [ count << count + 1 ].
```

Od předchozích případů se tyto konstrukce liší tím, že příjemcem zpráv je přímo samotný blok kódu. Z toho důvodu tedy není nutno v době běhu ověřovat, jestli se na zásobníku nenachází cizí typ objektu. Proto jsou rozšířené instrukce pro optimalizaci této konstrukce mnohem jednodušší, nemusí řešit situaci se zasíláním původní zprávy.

Při této optimalizaci jsou použity následující instrukce. Všechny instrukce mají jen jeden parametr a to o kolik bytů se má skočit, pokud je splněna podmínka. Pokud podmínka splněna není, pokračuje vykonávání kódu na další instrukci.

- `JUMP_IFTRUE` – Pokud je na vrcholu zásobníku objekt true, je ze zásobníku odstraněn a je proveden skok.
- `JUMP_IFFALSE` – Pokud je na vrcholu zásobníku objekt false, je ze zásobníku odstraněn a je proveden skok.
- `JUMP_IFNOTTRUE` – Pokud na zásobníku není objekt true, je proveden skok. V opačném případě je objekt true odstraněn ze zásobníku a vykonávání kódu pokračuje na další instrukci.
- `JUMP_IFNOTFALSE` – Pokud na zásobníku není objekt false, je proveden skok. V opačném případě je objekt false odstraněn ze zásobníku a vykonávání kódu pokračuje na další instrukci.

8.2.6 Instrukce pro cyklus – Iterace přes kolekci

V základní knihovně objekty typu List, String, Set a Dictionary obsahují slot `foreach:`. Metody v tomto slotu očekávájí jako parametr closure s jedním parametrem. Po spuštění metody je postupně aktivována tato closure přes všechny prvky, které daná kolekce obsahuje.
Příklad:

```
// Vypíše na standardní výstup všechny prvky kolekce
collection foreach: [ :each | each println ].
```

Aby však optimalizace této konstrukce nebyla vázána pouze na objekty ze základní knihovny, byl přidán objekt Iterator. Proto může programátor vytvářet vlastní kolekce, které dokáží využít optimalizace `foreach:`. Stačí pouze vyplnit patřičnou funkci do

`ObjectExtension`. Pro konkrétní kolekci je možné vytvořit objekt Iterator. Ten pak poskytuje funkci „vyber další prvek“. Takto mohou instrukce `FOREACH` a `NEXTITER` projít přes všechny prvky kolekce, která je libovolného typu.

Při překladu je odeslání zprávy `foreach:` nahrazeno instrukcí `FOREACH`. Pak je přepsán kód z closure. Za tento kód je vložena instrukce `NEXTITER`.

Instrukce `FOREACH` má tři parametry: pozici pro skok za `NEXTITER`, číslo lokální proměnné a číslo codeblocku. Pozice pro skok a číslo codebloku slouží ke stejným účelům jako v předchozích instrukcích, tedy k simulaci chování bez optimalizace. Na rozdíl například od instrukcí pro podmíněný skok neočekává na zásobníku instrukce `FOREACH` konkrétní objekt nebo konkrétní typ objektu. Aby byla splněna podmínka pro vykonání optimalizované části, musí objekt mít v `ObjectExtension` vyplněnou funkci `iterator_create` a `iterator_next`.

Po zavolání `iterator_create` je vytvořen objekt Iterator. Z něj je vybrán první objekt kolekce, který je vložen do patřičné lokální proměnné. Následně je Iterator vložen na zásobník a je prováděn další kód. Až je proveden celý kód, který vznikl vložením z closure, následuje vykonání instrukce `NEXTITER`.

Tato instrukce má dva parametry: pozici pro skok na začátek vloženého kódu (tedy hned za instrukci `FOREACH`) a číslo lokální proměnné. Když se začne provádět instrukce `NEXTIER`, tak na vrcholu zásobníku leží objekt, který je výsledkem předchozího výpočtu a pod ním leží objekt Iterator. Pokud Iterator vrátí další objekt, pak je objekt na vrcholu zásobníku odstraněn. Nový objekt, který je vrácen Iteratorem, je přiřazen do lokálního slotu. Poté se provede skok a tím další cyklus výpočtu closure pro další prvek kolekce.

Pokud Iterator již nevrátí další objekt, je ze zásobníku odstraněn (objekt, který je na vrcholu zásobníku, tam zůstává). Tím je cyklus dokončen a VM dále pokračuje ve vykonávání instrukcí za `NEXTITER`.

8.2.7 Výsledky výkonnostních testů

Výsledky výkonnostních testů pro porovnání přínosu rozšířených instrukcí je možné nalézt v tabulce 8.

Benchmark	Čas 1 (ms)	Čas 2 (ms)	Rozdíl	Rozdíl v %
bigobject	712	368	-344	-48,314%
gc	2731	2753	22	0,805%
mdispatch	876	626	-250	-28,538%
t1_recursive	504	305	-199	-39,484%
t1_sort	807	809	2	0,247%
t1_sum	170	141	-29	-17,058%
ccounter	91	40	-51	-56,043%
graph	95	81	-14	-14,736%
wcounter	304	304	0	0%
world	1648	1329	-319	-19,356%
sgfplayer	423	390	-33	-7,801%

Tabulka 8: Výsledky výkonnostních testů verze 0.12.0 s vypnutými (Čas 1) a povolenými (Čas 2) rozšířenými instrukcemi.

9 JIT kompilace

JIT (Just-In-Time) kompilace, nebo také dynamická kompilace, je častý způsob optimalizace používaný v jazycích s virtuálním strojem. Základním principem JIT kompilace je provádění překladu do nativního strojového kódu procesoru až v době běhu programu. JIT kompilace tedy nepřináší z pohledu programátora žádné nové vlastnosti, „pouze" je urychleno vykonávání programu.

Tato optimalizace se snaží získat výhody jak kompilovaného tak interpretovaného kódu. Jde především o tyto výhody:

- Kód zkompilovaný do nativního kódu procesoru běží řádově rychleji než interpretovaný kód. Překlad v době vykonávání programu má navíc výhodu zisku běhových informací o programu. Tyto informace při normálním překladu před spuštěním nejsou dostupné. Jedná se jednak o statistické informace o běhu programu a chování v určitých místech, ale také informace o možnostech konkrétního hardwaru, na kterém je program prováděn. Například možnost zjištění dostupnosti rozšiřujících instrukčních sad procesoru.

- Interpretovaný kód bývá většinou abstraktnější a přenositelnější. Není často tolik svázán s konkrétní platformou.

Hlavní nevýhodou JIT kompilace je, že samotný překlad v době běhu spotřebovává čas a zdroje, které by jinak byly využity pro samotné provádění programu. Proti této nevýhodě je většinou postupováno tak, že není překládán celý program. Jsou detekována a překládána pouze místa, která jsou v programu často využívána. Prostředky spojené s jejich kompilací se tedy díky efektivnějšímu vykonávání kritických míst rychle vrátí.

První nápady pro realizaci JIT kompilace se objevovaly od šedesátých let[8]. Mezi první jazyky, které ve svých implementacích podporovaly základy JIT kompilace, patří například: Lisp, FORTRAN, BASIC. K jazykům, které posunuly tuto technologii, také patří Smalltalk a Self[35].

Obzvláště Self patří mezi jazyky, které využívají JIT překlad velmi intenzivně. Self byl primárně experimentálním jazykem. Sun Microsystems pak využil poznatky získané jeho vývojem při práci na technologii HotSpot pro jazyk Java[9].

9.1 Knihovny pro JIT kompilaci

Protože by bylo velmi obtížné implementovat a udržovat vlastní knihovnu pro generátor nativního kódu pro různé architektury procesorů, rozhodl jsem se použít existující řešení. Dále v textu následuje stručný popis tří knihoven, které jsem zvažoval při výběru.

[35] Dva jazyky, ze kterých mj. vychází Kreatrix.

9.1.1 Knihovna libjit

Knihovna libjit je knihovna primárně zaměřená pro JIT kompilaci ve virtuálních strojích programovacích jazyků. Vznikla jako součást projektu DotGNU Portable.NET[36]. Knihovna je však plně nezávislá na tomto projektu a může být použita samostatně.

Cílem knihovny je poskytnout funkce pro pokrytí problematiky JIT kompilace s maximální univerzálností, bez svázání k nějakému konkrétnímu jazyku. Libjit neposkytuje kompletní virtuální stroj jako například LLVM (viz dále).

Knihovna je implementována v jazyce C a poskytována pod licencí GPL ve verzi 2[37]. Ve verzi 0.10.0 umožňuje generovat nativní kód pro tyto architektury: x86, x86-64, ARM a Alpha. Pro architektury, které nejsou přímo podporovány, je použit interní interpreter.

Základem fungování libjit je na platformě nezávislý tříadresový kód, který je sestavován voláním patřičných funkcí. K dispozici je „neomezený“ počet proměnných, které pak při kompilaci knihovna sama mapuje na registry konkrétního procesoru. API poskytuje základní funkce jako manipulace s obsahem proměnných, základní početní operace, volání funkcí, atp. Poskytuje však také složitější konstrukce, jako například možnost vytvořit vnořené funkce, které mají možnost pracovat s proměnnými uvnitř rodičovské funkce[38].

Po sestavení tříadresového kódu funkce je možné nechat vygenerovat nativní kód procesoru. K dispozici jsou dvě možnosti: buď okamžitá kompilace, nebo „líný“ přistup, kdy je provedena kompilace až po prvním zavolání dané funkce. Je také možné při kompilaci nastavovat úroveň optimalizace výsledného kódu.

Dokumentace ke knihovně není příliš bohatá, ale je celkem dostačující. K dispozici je také základní tutoriál a ukázková implementace Pascalu, která plně využívá libjit. API je velmi dobře navrženo, pojmenování funkcí je přehledné. Po naučení základních principů je pak používání knihovny bezproblémové. Dokumentace obsahuje také návod, jak přidat podporu pro další architekturu.

API poskytuje také několik funkcí pro ladění kódu. Je možné nechat vypsat výsledný tvar funkce v tříadresovém kódu nebo vypsat zkompilovaný tvar funkce v jazyce symbolických adres (syntaxe AT&T).

Původní domovská stránka `http://www.southern-storm.com.au` již pravděpodobně zanikla. Stažení nových verzí a online dokumentace je k nalezení například zde: `http://demakov.com/projects/index.html#libjit`.

Emailová konference dotgnu-libjit@gnu.org není příliš aktivní. Podle ní soudím, že projekt se již aktivně nevyvíjí a setrvává spíše ve fázi údržby. Reakce autorů na nalezené chyby a přijímání patchů je však rychlá. Z konference lze také vidět, že knihovna je používána i mimo projekt DotGNU Portable.NET.

[36] DotGNU Portable.NET je implementace Common Language Infrastructure (CLI) známějšího pod názvem „.NET“. Více informací o tomto projektu je možné nalézt na stránkách `http://dotgnu.org/`

[37] `http://www.gnu.org/licenses/old-licenses/gpl-2.0.html`

[38] Toto chování je například použito v jazyce Pascal.

9.1.2 Knihovna GNU lightning

Knihovna GNU lightning má podobné cíle jako knihovna libjit. Vytvořit abstraktní vrstvu pro generování nativního kódu vhodnou pro tvorbu JIT překladače. Od libjit se však v mnohém liší. GNU lightning je vůči libjit relativně malá a „odlehčená“ knihovna. Ve verzi 1.2 jsou podporovány tyto architektury: x86, PowerPC a SPARC.

GNU lightning z pohledu programátora-uživatele vytváří abstraktní RISC procesor. Tento procesor má 6 obecných registrů a 6 registrů pro uchování čísel v plovoucí řadové čárce. Dostupné instrukce jsou pak běžné instrukce RISC procesorů.

Při generování nativního kódu se snaží knihovna spotřebovat minimum prostředků, generování je poměrně rychlé a paměťově nenáročné. Způsob vytváření abstraktního kódu je podobný jako v případě libjit.

Knihovna má však díky své jednoduchosti několik nevýhod. Mezikód, který je vstupem pro knihovnu, má omezený počet registrů, takže rozvržení dat mezi ně musí řešit uživatel knihovny sám. Knihovna dále neprovádí žádné další optimalizace, většinou pouze přemapuje virtuální instrukce na skutečné pro daný procesor.

GNU lightning dále neposkytuje žádné nástroje pro ladění výsledného kódu. Jedinou možností tak zůstává přímo vygenerovaný kód, což ovšem vyžaduje znalost instrukční sady procesoru, pro který byl kód vygenerován.

Dokumentace není příliš rozsáhlá, ale vzhledem k rozsahu knihovny je dostačující. Obsahuje několik ukázkových příkladů, na kterých je demonstrováno generování kódu. Popsán je také způsob, jak přidat podporu pro další architekturu procesoru.

Knihovna je implementována v jazyce C a poskytována pod licencí GPL verze 2. Domovskou stránku projektu je možné nalézt zde: `http://www.gnu.org/software/lightning/`. Poslední verze, pro kterou je uvolněn balíček, je verze 1.2 z roku 2004. Emailová konference je však stále poměrně živá a v repositáři projektu v současné době (léto 2008) stále přibývají patche.

Projekty, které využívá GNU lightning, jsou například GNU Smalltalk[39] a GNU CLISP[40]. Existuje také projekt froofyJIT[41]. Tento projekt slouží jako C++ obal pro GNU lightning a přináší další možnosti v zápisu virtuálního kódu.

Přidat tuto knihovnu k vlastnímu programu je velmi jednoduché, pokud projekt používá Autotools[42]. Slouží k tomu skript `lightningize`, který zařídí většinu práce. Pak pouze stačí přidat jeden řádek do `configure.ac`. Po tomto nastavení se pokusí `configure` při instalaci najít GNU lightning v systému, Pokud není knihovna nalezena, použije se ta, která je přibalena k programu.

[39] `http://smalltalk.gnu.org/`
[40] `http://www.gnu.org/software/clisp/`
[41] `http://sourceforge.net/projects/froofyjit/`
[42] `http://airs.com/ian/configure/`

9.1.3 LLVM

LLVM[43] je kolekce nástrojů a knihoven, zahrnující překladač a instrukční sadu virtuálního RISC procesoru. LLVM je velký projekt a možnostmi daleko přesahuje „pouhou" JIT knihovnu. Knihovna dokáže generovat také statický kód. Možnosti použití se pohybují od kompilování plně statických jazyků po tvorbu abstraktních virtuálních strojů pro dynamické jazyky.

LLVM dokáže generovat statický kód pro celou řadu architektur. JIT překlad je možný pro x86, x86-64 a PowerPC.

Knihovny a nástroje LLVM jsou napsány v C++. Knihovna je uvolněna pod licencí University of Illinois Open Source License[44]. Tato licence je certifikovanou open source licencí[45].

LLVM je poměrně velký nástroj a uvádím ji zde hlavně jako alternativu. Pro potřeby JIT kompilace v Kreatrix je příliš komplexní a bylo by obtížné ji integrovat do současné implementace.

9.2 JIT kompilace

Z výše uvedených knihoven jsem si vybral pro implementaci v Kreatrix knihovnu libjit a to především z důvodu snadnějšího vytváření mezikódu, protože není omezena na fixní počet registrů. Dalším argumentem pak bylo mnohem snazší ladění. Tyto vlastnosti převážily nad GNU lightning, přestože bych jinak preferoval menší a jednodušší knihovnu.

9.2.1 Generování mezikódu

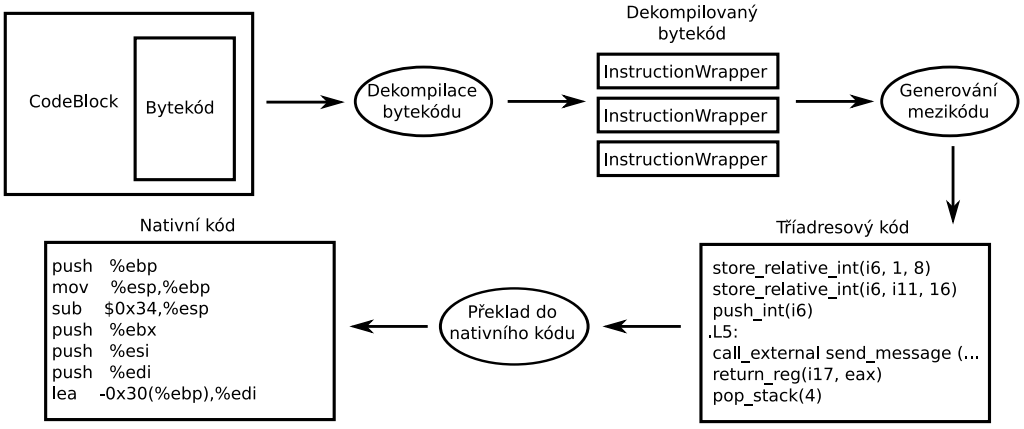
Prvním krokem k získání metody, která je zakompilována do podoby nativního kódu, je dekompilace jejího bytekódu. Dekompilovaný bytekód je reprezentován seznamem struktur `KxInstructionWrapper`. Tento seznam je vstupem pro generování mezikódu, který pak slouží jako vstup pro knihovnu libjit. Celý postup od bytekódu k JIT-kompilované metodě zachycuje obrázek 4. Příklad kódu, který je generován mezi jednotlivými fázemi, je v příloze A.

Mezikód je tříadresový kód. To znamená, že každá instrukce je popsána jménem instrukce, maximálně dvěma vstupními parametry a adresou pro uložení výsledku. Tento kód je pak převeden pomocí `libjit` do nativního kódu procesoru.

Generování tříadresového kódu je prováděno postupným průchodem přes instrukce bytekódu. Při tomto průchodu jsou navíc získávány informace o stavu zásobníku virtuálního stroje. Pomocí této informace je pak možné detekovat některá zbytečná zvedání čítače referencí pro objekty procházející zásobníkem.

Dále je možné pro jednoduché metody vynechat operace spojené s vytvořením a zrušením aktivačního záznamu.

[43] `http://llvm.org/`
[44] `http://llvm.org/svn/llvm-project/llvm/trunk/LICENSE.TXT`
[45] `http://www.opensource.org/licenses/UoI-NCSA.php`



Obrázek 4: Postup JIT kompilace

9.3 Omezení současné implementace JIT

Jeden z problémů použití libjit je způsoben tím, že metody v Kreatrix jsou obyčejné objekty, které mohou být odstraněny. Libjit ale neumí uvolnit paměť přidělenou již jednou zakompilované metodě. Tento problém není nijak zásadní, metody v běžném programu zas tak často nezanikají. Přesto ale v některých situacích může dojít k potížím. Například při způsobu vývoje, který je častý ve Smalltalkových prostředích, kdy je program vytvářen za chodu.

Další omezení současné implementace pramení z toho, že vznikla převážně jako proof-of-concept. Implementace není úplně kompletní, například nedokáže zkompilovat složitější metody. Narozdíl od ostatních optimalizací v této práci nemám v plánu JIT kompilaci prozatím začlenit do hlavního stromu Kreatrix.

Současná verze však ukázala, že vytvoření JIT je možné a může pomoci v návrhu dalších optimalizací do budoucna.

9.4 Výkonnostní srovnání při použití JIT

V tabulce 9 jsou uvedeny výsledky výkonnostních testů větvě s JIT překladačem vůči verzi 0.11.0. Větev s JIT kompilací byla původně založena na verzi 0.10.0 a později sloučena s verzí 0.11.0. Verze 0.12.0 už obsahuje příliš velké změny, aby bylo možné současnou podobu JIT jednoduše používat. Z toho důvodu je proveden test vůči verzi 0.11.0.

Test s JIT kompilací probíhal tak, že byly na počátku všechny metody, u kterých je to možné, před měřeným úsekem testu JIT kompilovány. Při reálném nasazení by pak byla využívána detekce kritických míst (viz kapitola 7.5).

Výsledky JIT kompilace, myslím, vypadají velmi optimisticky. Přestože JIT překladač umí přeložit v současné implementaci jen poměrně jednoduché metody, jsou výsledky testů výrazné. V kombinaci s rozšiřujícími instrukcemi a inline cache by mohl být výsledný výkon velmi dobrý.

Benchmark	Čas 1 (ms)	Čas 2 (ms)	Rozdíl	Rozdíl v %
bigobject	2479	2077	-402	-16,216%
gc	1697	2389	692	40,777%
mdispatch	1225	911	-314	-25,632%
t1_recursive	783	732	-51	-6,513%
t1_sort	1048	428	-620	-59,160%
t1_sum	214	197	-17	-7,943%
ccounter	137	132	-5	-3,649%
graph	148	119	-29	-19,594%
wcounter	398	214	-184	-46,231%
world	2873	2538	-335	-11,660%
sgfplayer	695	578	-117	-16,834%

Tabulka 9: Výsledky výkonnostních testů při Kreatrix s JIT překladačem (Čas 2) vůči verzi 0.11.0 (Čas 1).

Přidat podporu JIT kompilace do změn ve verzi 0.12.0 však ještě vyžaduje vyřešit některé dílčí problémy. Navíc by to velmi ztížilo případné změny ve virtuálním stroji. Protože se chci vyhnout přílišnému svázání implementace, zůstává JIT kompilace nadále jako samostatná větev.

10 Profilování programů vytvořených v Kreatrix

Doposud byly obsahem této práce způsoby, jak obecně zefektivnit vykonávání kódu ve virtuálním stroji. Jednalo se víceméně o transparentní změny z pohledu programátora v Kreatrix. Pro efektivní vykonávání programu je ale také důležité, aby programátor, který se nezajímá o vnitřní stavbu Kreatrix, mohl jednoduše analyzovat svůj program a nalézt v něm výkonnostně problémová místa. Z tohoto důvodu jsem vytvořil modul `profiler`.

Existují dva základní přístupy k profileru: deterministický a statistický. Deterministický zaznamenává čas trvání pro každou provedenou funkci. Naopak statistický pouze „vzorkuje“ běh programu. Výhodou deterministického jsou přesnější výsledky. Nevýhodou je vysoká režie a z toho vyplývající zpomalení sledovaného programu. Statistický poskytuje méně přesná čísla, ale jeho dopad na běh programu je výrazně menší.

Modul `profiler` poskytuje deterministický profiler. Kreatrix je v současné podobě interpretovaným jazykem a tedy samotné vykonávání bytekódu má relativně velkou režii. Proto náklady spojené s během deterministického profileru nejsou příliš výrazné.

10.1 Použití profileru

Použití asi nejlépe demonstruje příklad:

```
import: "profiler".

runProfiler = { |stat|

    stat << profiler profileBlock: [
       doSomething. // Profilovaná operace
    ].

    stat sortByCumulativeTime.
    stat writeHtmlToFile: "output_ctime.html".
    stat sortByTime.
    stat writeHtmlToFile: "output_time.html".
}.
```

Blok předaný zprávě `profileBlock:` je proveden a při jeho provádění jsou sbírána profilovací data. Výsledek je pak zapsán do dvou HTML souborů. V každém souboru jsou data setříděna podle jiného kritéria.

Jak může vypadat HTML výstup, je možné vidět na obrázku 5. Všechny časy jsou uvedené v sekundách. V závorce je pak procentuální vyjádření času vůči celkovému času. Jak je vidět z obrázku, je zaznamenána celková doba běhu profilované části, kolikrát byl spuštěn garbage collector a jak dlouho trvalo jeho zpracování.

V tabulce je pak zaznamenán každý aktivovatelný objekt, který byl v době sledování spuštěn. Sloupce v tabulce obsahují následující informace: počet aktivování daného

Total time: 3.020368

GC was called 3x (cumulative time: 0.000463 (0.02%))

Sorted by: time

#	Count	Time	Cumulative time	Object message	Place
1	135515	0.364038 (12.05%)	0.652545 (21.60%)	Integer @	<vm_init>:136
2	55271	0.191452 (6.34%)	0.333979 (11.06%)	Point hash	<vm_init>:764
3	39903	0.143789 (4.76%)	0.244834 (8.11%)	Point ==	<vm_init>:756
4	64016	0.128733 (4.26%)	1.967005 (65.12%)	true ifTrue:	CFunction
5	18272	0.123220 (4.08%)	2.082197 (68.94%)	List foreach:	CFunction
6	144761	0.113548 (3.76%)	0.115723 (3.83%)	Object clone	CFunction
7	16820	0.108272 (3.58%)	1.038336 (34.38%)	Goban _fourPlacesAround:	test3/sgfplayer/goban.kx:131
8	3559	0.101022 (3.34%)	1.762896 (58.37%)	ScopedBlock whileFalse:	CFunction
9	33640	0.098478 (3.26%)	0.305069 (10.10%)	Point +	<vm_init>:774

Obrázek 5: Výřez HTML výstupu z modulu `profiler`

objektu (count), čas strávený při vykonávání objektu (time), celkový čas strávený prováděním objektu včetně zpracování zpráv, které daný objekt zaslal (cumulative time). Sloupec „Object message“ obsahuje jméno objektu, ve kterém se aktivovaný objekt nachází a zprávu, která způsobila aktivaci objektu. Poslední sloupec pak obsahuje informaci o umístění objektu: jméno souboru a číslo řádku, případně informaci, že se jedná o `CFunction`.

10.2 Implementace profileru

Základní nutností, kterou modul `profiler` potřebuje ke své činnosti, je informace o tom, že byl nějaký objekt aktivován. Každý aktivovatelný objekt má v `ObjectExtension` vyplněn ukazatel na funkci `activate`, viz podkapitola 3.1.3. Při aktivaci objektu je tato funkce zavolána.

Pokud je zahájeno profilování, nahradí profiler v aktivovatelných objektech tuto funkci za vlastní. Tato nová funkce pak zavolá tu původní. Navíc ale profiler získá informaci o spuštění a dokončení aktivace daného objektu.

Toto řešení má několik výhod. Aby mohl profiler fungovat, není třeba nijak speciálně upravovat či kompilovat virtuální stroj. Navíc je možné profilaci libovolně zapínat a vypínat, jak je potřeba. Při pozastaveném sledování pak není rychlost virtuálního stroje nijak ovlivněna profilerem.

Získaná data jsou ukládána do objektu `PTracker`. Pro každý aktivovatelný objekt, který byl v době sledování alespoň jednou aktivován, je `PTracker` vytvořen. Každý `PTracker` sleduje vždy jeden objekt. Do `PTrackeru` jsou zaznamenávány tyto údaje:

- `counter` – Počítadlo aktivací sledovaného objektu.
- `cumulative_time` – Celkový čas, který zabralo provádění aktivace objektu včetně času zpracování zpráv, které byly zaslány v rámci této aktivace.

- `children_time` – Čas, po který byl objekt aktivován, ale nebyl prováděn jeho vlastní kód. Čistý čas zpracování objevující se na výstupu profileru je rozdíl mezi `cumulative_time` a `children_time`.
- `is_running` – Příznak určující jestli je sledovaný objekt právě prováděn. Slouží k detekci rekurzivního volání.
- `tracked_messages` – Pole se jmény zpráv a vlastníky slotů, kteří způsobili spuštění objektu.

Když dojde k aktivaci objektu, jsou vykonány následující akce:

1. Z globální proměnné je získán aktuálně běžící `PTracker` (dále jako rodičovský `PTracker`).
2. Z aktivovaného objektu je získán `PTracker` (dále jako aktuální `PTracker`). Tento `PTracker` je nastaven jako aktuálně běžící.
3. Je zaznamenán aktuální čas.
4. Je spuštěna původní `activate`.
5. Po dokončení `activate` je změřen čas a doba běhu je připsána do aktuálního `PTrackeru` do položky `cumulative_time`. Do rodičovského `PTrackeru` je čas přičten do položky `children_time`. Pokud se ale jedná o rekurzivní volání, je v rodičovském `PTrackeru` naopak čas odečten.
6. Jako aktuálně bežící `PTracker` je vrácen původní rodičovský `PTracker`.

Aby byl `PTracker` co nejrychleji dosažitelný, je vkládán přímo do aktivovatelných objektů. V případě `CFunction` do pomocných objektů a v případě `CodeBlock` jako skrytý literál.

Pokud je v průběhu sledování spuštěn garbage collector, je jeho spuštění a čas zaznamenán. Doba běhu GC je pak přičtena do `children_time` naposledy aktivovaného objektu.

10.3 Další vlastnosti profileru

Při používání profileru je nutné brát ohled na omezenou přesnost hodin a také, že profiler přidává vlastní režii do programu. Tyto faktory mohou zkreslit výsledky u velmi krátkých metod.

Dále je také třeba brát v úvahu, že čas zpracování zpráv, ve kterých je výsledkem neaktivovatelný objekt, je přičítán objektu, který zprávu zaslal. Čas strávený vyhledáváním objektu ve slotech, i v případě aktivovatelných objektů, je také započítán zasilateli zprávy.

Současná implementace je omezena pouze na jednovláknové programy. Vytvářet vlákna se spuštěným profilerem je sice možné, ale výsledky profileru pak budou zkreslené. Do budoucna plánuji rozšířit podporu i pro vícevláknové programy zhruba stejným způsobem, jakým nyní funguje započítávání běhu GC.

11 Srovnání s implementacemi jiných jazyků

Cílem této kapitoly je provést srovnání rychlosti implementace Kreatrix s implementacemi podobných jazyků. Toto srovnání je však poměrně hrubé a výsledky jsou spíše orientační. Testy, na kterých byla měření prováděna, jsou celkem jednoduché a v rozsáhlejších programech se mohou výsledky lišit.

Pro maximální objektivnost jsem použil výkonnostní testy z projektu The Computer Language Benchmarks Game[46]. Do této soutěže může kdokoli zaslat svou vlastní verzi testu pro daný jazyk a tato verze bude zařazena, pokud bude lepší než dosavadní verze. Soutěž už běží několik let, a domnívám se tedy, že by nemělo dojít k chybným výsledkům z důvodu nekvalitní implementace.

11.1 Srovnávané jazyky

Ke srovnání jsem vybral implementace těchto jazyků: Python, Io, Ruby a PHP. Srovnávané jazyky jsem vybral podle toho, že mají podobné vlastnosti, cíle nebo rozsah potenciálního nasazení jako Kreatrix.

Všechny čtyři jazyky jsou dynamicky typované, interpretované a podporují objektově orientované programování. Stejně jako Kreatrix mají také virtuální stroj implementován v jazyku C.

Testy jsem spouštěl na standardních instalacích jednotlivých implementací bez použití urychlovacích nástrojů (jako například Psyco[47] pro Python). Kreatrix byla testována na verzi 0.12.0. Tato verze obsahuje všechny úspěšné optimalizace popsané v této práci, s výjimkou JIT kompilace. Větev s JIT kompilátorem není do 0.12.0 zahrnuta a ani při tomto srovnání nebyla použita.

11.1.1 Python

Python je interepretovaný, dynamický, objektově orientovaný jazyk. Ve své kategorii dnes jeden z nejpopulárnějších a je široce používán. Implementace Pythonu mi byla inspirací při práci na Kreatrix.

Python navrhl v roce 1990 Guido van Rossum. Domovskou stránku projektu je možné nalézt na adrese `http://www.python.org`. Výkonnostní test byl prováděn na verzi 2.5.2.

11.1.2 Io

Io je jazyk založený na prototypech, převážně inspirován Smalltalkem, Selfem, NewtonScriptem a Lispem. Tento jazyk byl jedním z hlavních vzorů při návrhu Kreatrix. Io není příliš známým jazykem, přestože obsahuje spoustu zajímavých nápadů. Jedno z nejznámějších nasazení v praxi je ve společnosti Pixar Animation Studios[48].

[46] `http://shootout.alioth.debian.org/`
[47] `http://psyco.sourceforge.net/`
[48] `http://www.iolanguage.com/blog/blog.cgi?do=item&id=104`

Autorem Io je Steve Dekorte, který začal pracovat na projektu v roce 2002. Domovská stránka projektu se nalézá na adrese: `http://iolanguage.com/`. Pro testování jsem použil verzi 20080120 z Git repositáře projektu.

11.1.3 Ruby

Ruby je další ze zástupců dynamických objektových jazyků. Je inspirován především jazyky Perl a Smalltalk. Nejznámější použití je asi ve webovém frameworku Ruby on Rails[49]. Autorem jazyka je Yukihiro Matsumoto. Projekt sídlí na adrese: `http://www.ruby-lang.org/`. Pro testování jsem použil verzi 1.8.6.

11.1.4 PHP

PHP je známo především jako skriptovací jazyk používaný pro tvorbu webových stránek. Objektově orientované programování se poprvé objevilo ve verzi 3. V dalších verzích však bylo značně změněno. Osobně nepovažuji návrh tohoto jazyka za příliš povedený. PHP je však používáno ve velkém měřítku, takže pro účely tohoto srovnávacího testu se hodí.

Projekt byl zahájen v roce 1994 a jeho původním autorem byl Rasmus Lerdorf. Stránky projektu je možné nalézt na adrese: `http://php.net/`. Pro testování jsem použil verzi 5.2.6.

11.2 Testy

Vybral jsem 4 testy, na kterých jsem provedl měření. Odkazy na kompletní zadání testu uvádím jako poznámky pod čarou. Jedná se o tyto testy:

- *recursive* – Tento test provede naivním rekurzivním způsobem výpočet tří funkcí: Ackermanovu funkci, Fibbbonačiho funkci a TAK funkci.[50]
- *n-body* – Modelování orbity několika planet.[51]
- *k-nucleotid* – Test načte řetězec DNA a pak provádí vyhledávání vzorků podle velikosti.[52]
- *nsieve* – Test generuje prvočísla pomocí metody Erastetanova síta.[53]

[49] `http://www.rubyonrails.org/`

[50] `http://shootout.alioth.debian.org/gp4/benchmark.php?test=recursive\&lang=all\#about`

[51] `http://shootout.alioth.debian.org/debian/benchmark.php?test=nbody\&lang=all\#about`

[52] `http://shootout.alioth.debian.org/debian/benchmark.php?test=knucleotide\&lang=all\#about`

[53] `http://shootout.alioth.debian.org/gp4/benchmark.php?test=nsieve\&lang=all\#about`

Jazyk	Recursive	N-body	Nsieve	K-nucleotid
Kreatrix	8.34	13.52	46.26	0.745
Python	8.22	8.26	8.85	0.218
Ruby	16.45	17.68	68.87	0.621
PHP	11.82	10.02	23.95	0.408
Io	69.55	14.50	9665	Není k dispozici

Tabulka 10: Výsledky srovnávacích testů. Číslo označuje délku běhu testu v sekundách.

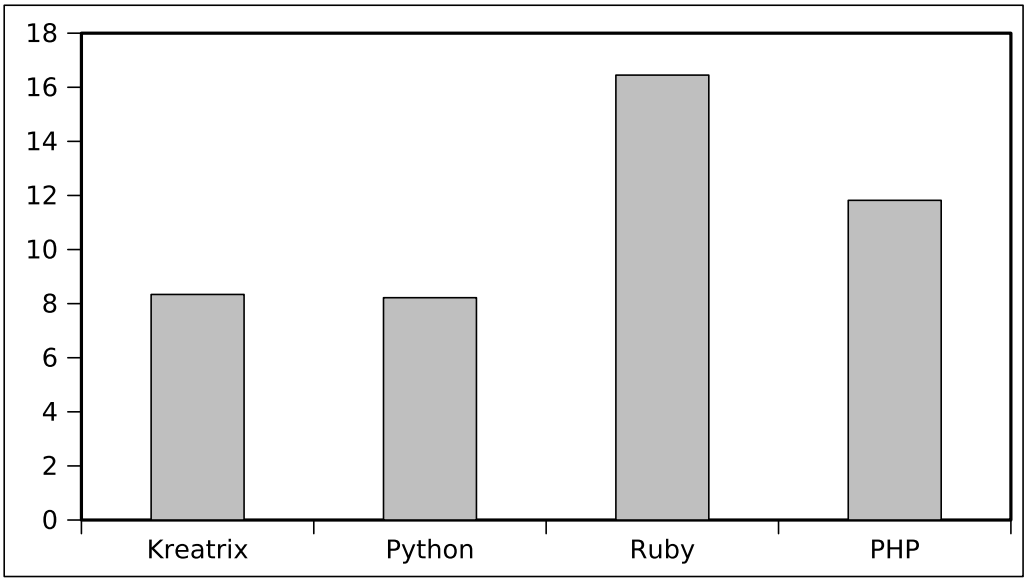
11.3 Porovnání výsledků výkonnostních testů

V tabulce 10 jsou uvedeny doby běhu jednotlivých testů. Na obrazcích 6, 7, 8 a 9 jsou pak tytéž hodnoty zobrazeny jako sloupcové grafy.

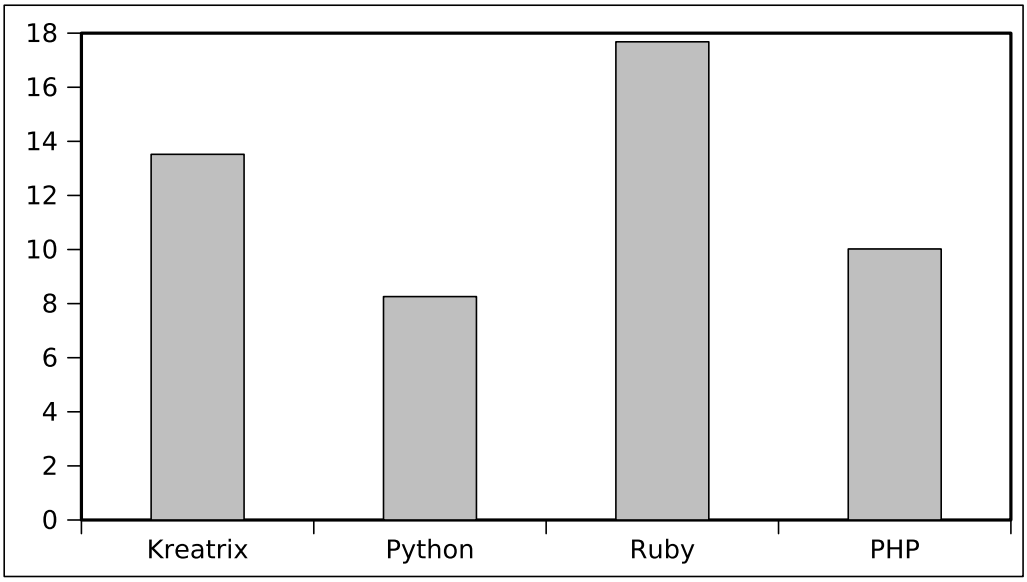
Test *recursive* byl spuštěn s parametrem 7, *n-body* s parameterem 200000 a *nsieve* s parametrem 11. Test *k-nucleotid* byl prováděn nad souborem s ukázkovým vstupem.

Přestože se jedná jen o velice hrubé porovnání, myslím, že tento test napovídá, že Kreatrix může být s těmito jazyky po výkonnostní stránce poměrně dobře srovnávána.

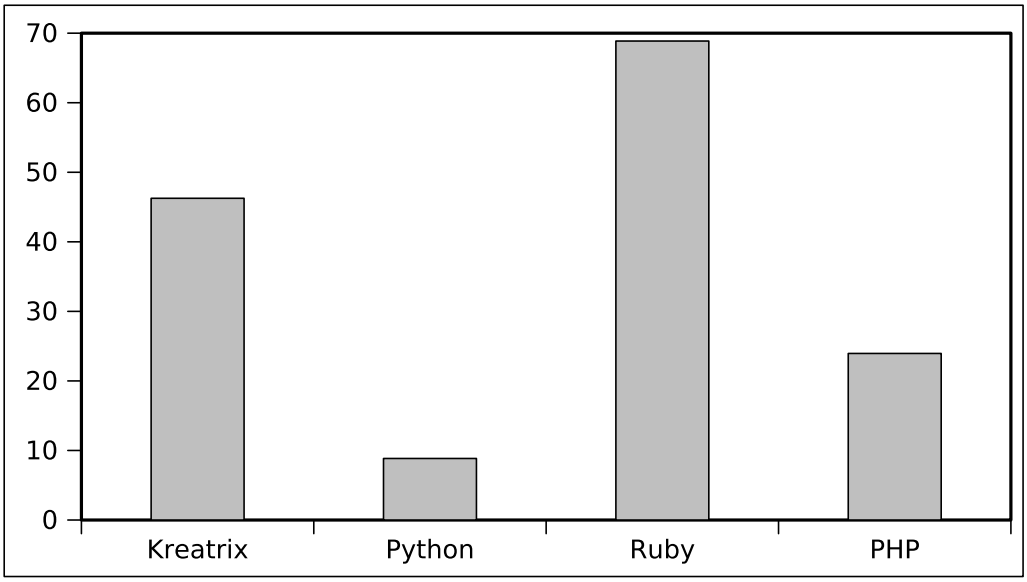
Slabinu Kreatrix vidím hlavně v implementaci základní knihovny, která nedosahuje takové rychlosti jako u konkurentů. Toto se nejvíce projevuje v test *k-nucleotid*. Naopak samotná interpretace bytekódu je na srovnatelné úrovni.



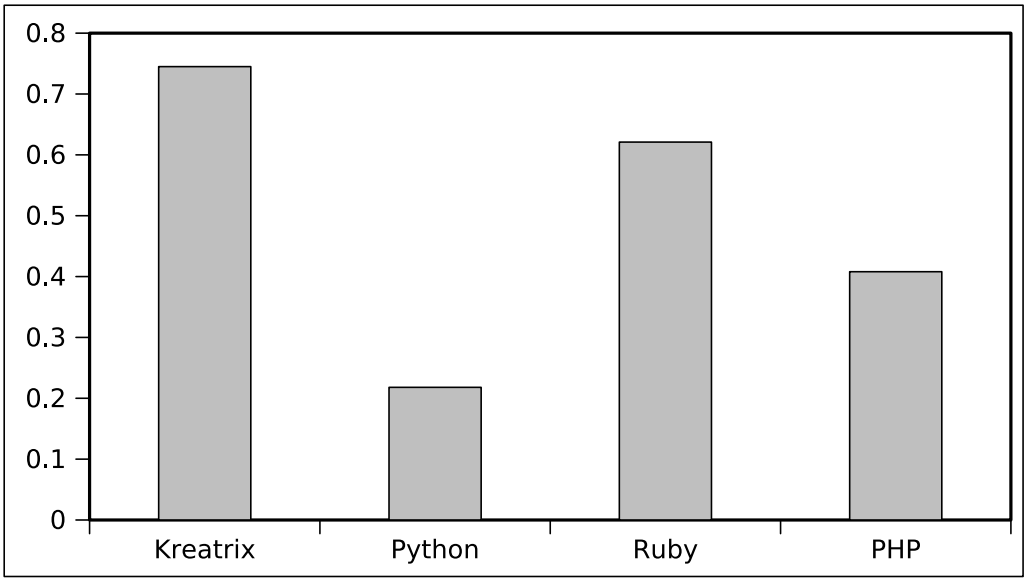
Obrázek 6: Graf doby běhu testu *recursive*



Obrázek 7: Graf doby běhu testu *n-body*



Obrázek 8: Graf doby běhu testu *nsieve*



Obrázek 9: Graf doby běhu testu *k-nucleotid*

12 Návrh dalšího postupu při optimalizování Kreatrix

Výsledky v předchozí kapitole naznačují, že Kreatrix se svým výkonem blíží koncepčně podobným jazykům. Tyto jazyky se už vyvíjejí poměrně dlouho a nepředpokládám radikální skok v jejich výkonu. Z toho také usuzuji, že se Kreatrix se svým současným konceptem virtuálního stroje blíží k výkonnostnímu limitu. Z tohoto důvodu vidím jako možné pokračování jednu z následujících cest.

Nejjednodušší cestou je optimalizace pro konkrétní případy, kdy budou například přepsány do C některé části základní knihovny, případně budou uvedeny nové rozšiřující instrukce. Praktickým příkladem by mohlo být vytvoření nového objektu Directory (asociativní pole), který by byl specializován pouze na klíče z řetězců.

Výhodou tohoto přístupu je relativní jednoduchost implementace bez nutnosti měnit jádro virtuálního stroje. Hlavní nevýhoda spočívá v tom, že takovéto optimalizace vedou ke zrychlení jen pro specifické problémy a nejedná se o plošné zlepšení.

Druhá možnost je provádět komplexní analýzu kódu, což by spolu s JIT překladačem mohlo přinést razantní urychlení. Bylo by ale nutné provést některé změny v implementaci a virtuální stroj by se stal mnohem složitějším a byl by hůře přenositelný. Také by bylo nutné vyřešit problémy, jak zachovat poměrně velkou dynamičnost jazyka.

Třetí možností je pak upustit od současného virtuálního stroje a použít existující projekt specializující se tímto směrem. V úvahu přicházejí například JVM, LLVM nebo Parrot[54]. Obávám se však, že přenést sémantiku Kreatrix na tyto virtuální stroje, může přinést mnohé problémy, vzhledem k jinému pojetí objektů, než jaké nabízí většina ostatních jazyků. Nejvíce nadějí vkládám do Parrotu, který si klade za cíl být virtuálním strojem pro všechny dynamické jazyky. Zatím je ale tento projekt stále pod intenzivním vývojem.

[54] `http://www.parrot.org/`

13 Závěr

Účelem této práce bylo zlepšit výkonnost implementace jazyka Kreatrix. Domnívám se, že tohoto cíle bylo dosaženo. V tabulce 11 je vidět srovnání výkonnostních testů verzí před a po této práci. Na obrázku 10 jsou v grafu vyneseny časy dvou nejsložitějších testů.

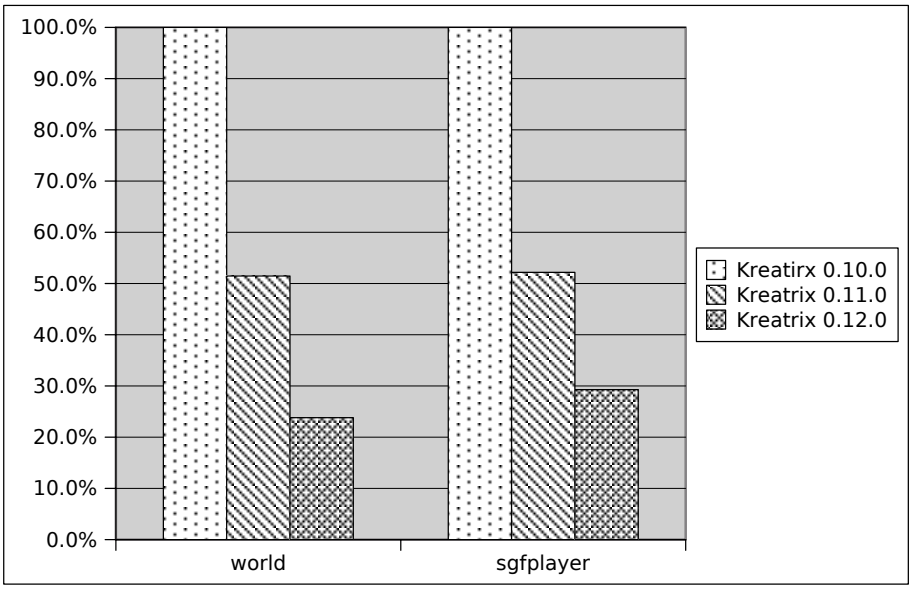
Z údajů je patrné, že virtuální stroj Kreatrix byl zrychlen zhruba čtyřikrát. Porovnání s implementacemi podobných jazyků dopadlo také poměrně dobře. Z těchto důvodů tedy věřím, že tato práce představuje důležitý krok k větší praktické použitelnosti celého projektu.

Tato práce pro mne také znamenala spoustu zkušeností s profilováním a optimalizacemi. Vyzkoušel jsem si několikrát, že v těchto problémech může být intuice zrádná a je mnohdy těžké odhadnout skutečný přínos různých optimalizací a vždy je potřeba důkladného měření.

Po skončení této práce budu samozřejmě na projektu nadále pracovat.

Benchmark	Čas 1 (ms)	Čas 2 (ms)	Rozdíl	Rozdíl v %
gc	5519	2753	-2766	-50,117%
mdispatch	2623	626	-1997	-76,134%
t1_recursive	1767	305	-1462	-82,739%
t1_sort	2724	809	-1915	-70,301%
t1_sum	543	141	-402	-74,033%
ccounter	291	40	-251	-86,254%
graph	314	81	-233	-74,203%
wcounter	856	304	-552	-64,485%
world	5582	1329	-4253	-76,191%
sgfplayer	1332	390	-942	-70,720%

Tabulka 11: Porovnání doby běhu výkonnostních testů mezi verzí 0.10.1 a 0.12.0



Obrázek 10: Srovnání doby běhu testu *world* a *sgfplayer* mezi verzemi 0.10.0 a 0.12.0

14 Reference

[1] Böhm, S.: Implementace programovacího jazyka pro MMORPG Dungeon Oline (2007)

[2] Dean, J.A.: Abstract whole-program optimization of object-oriented languages (1996)

[3] Hölzle, U., Chambers, C., Ungar, D.: Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In: ECOOP '91: Proceedings of the European Conference on Object-Oriented Programming, London, UK, Springer-Verlag (1991) 21–38

[4] Chambers, C., Ungar, D., Lee, E.: An efficient implementation of self a dynamically-typed object-oriented language based on prototypes. SIGPLAN Not. **24**(10) (1989) 49–70

[5] Driesen, K., Hölzle, U.: The direct cost of virtual function calls in c++. In: OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, New York, NY, USA, ACM (1996) 306–323

[6] R. Pagh, F.R.: Cuckoo hashing. (2001)

[7] Sedgewick, R.: Algoritmy v C, části 1-4. Softpress (2003)

[8] Aycock, J.: A brief history of just-in-time. ACM Comput. Surv. **35**(2) (2003) 97–113

[9] Russell, K., Bak, L.: The hotspottm serviceability agent: an out-of-process high level debugger for a javatm virtual machine. In: JVM'01: Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium, Berkeley, CA, USA, USENIX Association (2001) 16–16

[10] Hölzle, U., Agesen, O.: (Dynamic vs. static optimization techniques for object-oriented languages)

[11] Goldberg, A., Robson, D.: Smalltalk-80: The Language and its Implementation. Addison-Wesley (1983)

A Ukázka generování kódu při JIT kompilaci

Původní kód:

```
method = {
    "Hello world!" println.
}.
```

Dekompilovaný bytekód:

```
push_literal 0
send_unary 0
pop
return_self
```

Tříadresový kód:

```
function FCN(i1 : ptr, i2 : ptr) : ptr
incoming_frame_posn(i1, 8)
incoming_frame_posn(i2, 12)
i6 = &s5
i8 = load_relative_int(i2, 0)
store_relative_int(i6, 134793016, 0)
store_relative_int(i6, 134746992, 4)
store_relative_int(i6, 0, 8)
push_int(i6)
.L2:
call_external send_message (0x08054ac0)
return_reg(i14, eax)
pop_stack(4)
if itrue(i14) then goto .L0
.L3:
push_int(0)
push_int(i6)
push_int(134796856)
.L4:
call_external returned_null_value (0x08060750)
return_int(0)
ends_in_dead
.L5:
pop_stack(12)
.L0:
i19 = load_relative_int(i14, 0)
i21 = i19 - 1
store_relative_int(i14, i21, 0)
if itrue(i21) then goto .L1
```

```
.L6:
push_int(i14)
.L7:
call_external free_object (0x0804fe10)
pop_stack(4)
.L1:
i24 = load_relative_int(i8, 0)
i26 = i24 + 1
store_relative_int(i8, i26, 0)
return_int(i8)
ends_in_dead
end
```

Výsledný nativní kód:

```
b7f75017 <.text>:
b7f75017: 55                    push   %ebp
b7f75018: 8b ec                 mov    %esp,%ebp
b7f7501a: 83 ec 30              sub    $0x30,%esp
b7f7501d: 53                    push   %ebx
b7f7501e: 56                    push   %esi
b7f7501f: 57                    push   %edi
b7f75020: 8d 7d d0              lea    -0x30(%ebp),%edi
b7f75023: 8b 45 0c              mov    0xc(%ebp),%eax
b7f75026: 8b 18                 mov    (%eax),%ebx
b7f75028: c7 07 38 c7 08 08     movl   $0x808c738,(%edi)
b7f7502e: c7 47 04 70 13 08 08  movl   $0x8081370,0x4(%edi)
b7f75035: c7 47 08 00 00 00 00  movl   $0x0,0x8(%edi)
b7f7503c: 57                    push   %edi
b7f7503d: e8 7e fa 0d 50        call   0x8054ac0
b7f75042: 83 c4 04              add    $0x4,%esp
b7f75045: 8b f0                 mov    %eax,%esi
b7f75047: 0b c0                 or     %eax,%eax
b7f75049: 0f 85 14 00 00 00     jne    0xb7f75063
b7f7504f: 6a 00                 push   $0x0
b7f75051: 57                    push   %edi
b7f75052: 68 38 d6 08 08        push   $0x808d638
b7f75057: e8 f4 b6 0e 50        call   0x8060750
b7f7505c: 33 c0                 xor    %eax,%eax
b7f7505e: e9 25 00 00 00        jmp    0xb7f75088
b7f75063: 8b 06                 mov    (%esi),%eax
b7f75065: 2d 01 00 00 00        sub    $0x1,%eax
b7f7506a: 89 06                 mov    %eax,(%esi)
b7f7506c: 0b c0                 or     %eax,%eax
b7f7506e: 0f 85 09 00 00 00     jne    0xb7f7507d
```

```
b7f75074: 56                    push   %esi
b7f75075: e8 96 ad 0d 50        call   0x804fe10
b7f7507a: 83 c4 04              add    $0x4,%esp
b7f7507d: 8b 03                 mov    (%ebx),%eax
b7f7507f: 05 01 00 00 00        add    $0x1,%eax
b7f75084: 89 03                 mov    %eax,(%ebx)
b7f75086: 8b c3                 mov    %ebx,%eax
b7f75088: 8b 5d cc              mov    -0x34(%ebp),%ebx
b7f7508b: 8b 75 c8              mov    -0x38(%ebp),%esi
b7f7508e: 8b 7d c4              mov    -0x3c(%ebp),%edi
b7f75091: 8b e5                 mov    %ebp,%esp
b7f75093: 5d                    pop    %ebp
b7f75094: c3                    ret
end
```

B Obsah CD

B.1 Adresářová struktura

Přiložené CD obsahuje tyto adresáře:

- `kreatrix` – Adresář obsahuje hlavní větev Kreatrix ve verzi 0.12.0.
- `kreatrix-jit` – Adresář obsahuje větev Kreatrix s JIT překladačem.
- `kreatrix-sc` – Adresář s větví projektu s alternativní implementací profilů
- `shootout` – Zdrojové kódy testů, které byly použity pro porovnání rychlostí mezi jazyky.

Všechny větve projektu jsou plnohodnotnou větví nástroje Bazaar[55]. V každé větvi je tedy kompletně přístupná historie změn. Odkaz na aktuální vývojovou větev je možné nalézt na stránce: `http://www.kreatrix.org/download`. Dále se na CD nachází tato práce a instalační balíček Kreatrix-0.12.0.

B.2 Instalace

Zde je popsán postup, jak nainstalovat Kreatrix z distribučního balíku. Kompilaci lze ale provést přímo z adresáře `kreatrix`.

1. Rozbalit distribuční tarball.

   ```
   tar xvzf kreatrix-0.12.0.tar.gz
   ```

2. Provedení konfigurace. Ve většině případů stačí:

   ```
   ./configure
   ```

   Informace k dalšímu nastavení lze získat pomocí parametru `--help`.

3. Kompilace:

   ```
   make
   ```

4. Instalace (v základní konfiguraci vyžaduje práva super uživatele):

   ```
   make install
   ```

[55] `http://bazaar-vcs.org/`

B.3 Typické použití

- Překlad zdrojového souboru a jeho spuštění:

  ```
  kreatrix <název_souboru>
  ```

- Spuštění v interaktivním režimu:

  ```
  kreatrix -i
  ```